



AFRL-RY-WP-TR-2011-1029



EXTREME SCALE COMPUTING STUDIES

Mark A. Richards and Daniel P. Campbell

Georgia Institute of Technology

DECEMBER 2010

Final Report

Approved for public release; distribution unlimited.

See additional restrictions described on inside pages

STINFO COPY

**AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Defense Advanced Research Projects Agency's Public Release Center and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (<http://www.dtic.mil>).

AFRL-RY-WP-TR-2011-1029 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

*//Signature//

KERRY HILL, Program Manager
Advanced Sensor Components Branch
Aerospace Components & Subsystems Division

//Signature//

BRADLEY J. PAUL, Chief
Advanced Sensor Components Branch
Aerospace Components & Subsystems Division

//Signature//

JEFF HUGHES, Chief
Aerospace Components & Subsystems Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show “//Signature//” stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
<p>The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.</p>					
1. REPORT DATE (DD-MM-YY) December 2010		2. REPORT TYPE Final		3. DATES COVERED (From - To) 18 May 2007 – 31 December 2010	
4. TITLE AND SUBTITLE EXTREME SCALE COMPUTING STUDIES				5a. CONTRACT NUMBER FA8650-07-C-7724	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER 62303E	
6. AUTHOR(S) Mark A. Richards and Daniel P. Campbell				5d. PROJECT NUMBER ARPS	
				5e. TASK NUMBER ND	
				5f. WORK UNIT NUMBER ARPSNDBJ	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Georgia Institute of Technology A Unit of the University System of Georgia Georgia Tech Research Institute Atlanta, GA 30332				8. PERFORMING ORGANIZATION REPORT NUMBER 210667V	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory Sensors Directorate Wright-Patterson Air Force Base, OH 45433-7320 Air Force Materiel Command United States Air Force				10. SPONSORING/MONITORING AGENCY ACRONYM(S) AFRL/Rydi	
				11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S) AFRL-RY-WP-TR-2011-1029	
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.					
13. SUPPLEMENTARY NOTES DISTAR case 17642; Clearance Date: 18 July 2011. This report contains color.					
14. ABSTRACT Four studies were conducted to determine the technical developments needed for a 1,000× increase in computational capability by 2015: 1. The Exascale Computing Study, addressing hardware and system architecture issues; 2. The Exascale Computing Software Study, addressing software technologies to effectively utilize extreme levels of concurrency; 3. The Exascale Computing Resiliency Study, addressing practical fault management in extreme scale systems; and 4. The Embedded Extreme-scale System Study, addressing elements of extreme scale applications and benchmarks, low-power algorithms, design simulation methods, programmability metrics, and enabling tools for graphics processing unit use. The studies identified critical technology challenges in power and energy; concurrency; resiliency; and memory and storage. Investments were recommended in locality and parallelism expression and optimization; power management; self-awareness techniques; execution models; extreme scale benchmarks, programmability metrics, low power algorithms, and libraries for graphics processing units.					
15. SUBJECT TERMS exascale, extreme scale, high performance computing, graphics processing unit, GPU, VSIPL, locality, parallelism, parallel computing, benchmarking, benchmarks, concurrency, self-aware computing, resiliency, checkpointing, Moore's Law, programmability, execution model					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT: SAR	18. NUMBER OF PAGES 590	19a. NAME OF RESPONSIBLE PERSON (Monitor) Kerry Hill 19b. TELEPHONE NUMBER (Include Area Code) N/A
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified			

TABLE OF CONTENTS

<u>Section</u>	<u>Page</u>
List of Tables	iii
Preface.....	iv
Acknowledgements	vii
1. Summary	1
1.1 Goals of the Project.....	1
1.2 Project Approach	1
1.3 Principal Results and Conclusions	1
1.4 Major Recommendations.....	2
2. Introduction	5
2.1 Goal of the Project.....	5
2.2 Extreme Scale Computing Systems.....	5
2.3 DARPA's Extreme Scale Computing Studies	5
2.4 Embedded Extreme Scale Study.....	6
2.5 Organization of This Report.....	6
3. Methods and Procedures	7
3.1 Group Studies.....	7
3.2 Embedded Extreme Scale Studies	9
4. Results and Discussion.....	11
4.1 Group Studies.....	11
4.2 Embedded Extreme Scale Study	13
4.2.1 Design Environments for Terascale Embedded Computing.....	13
4.2.2 Extreme Scale Embedded Computing Applications.....	14
4.2.3 Low Power Computational Algorithms	14
4.2.4 High Performance Libraries for Advanced Graphics Processing Units	15
4.2.5 Metrics for Extreme Scale Systems	15
4.3 Discussion.....	16
5. Conclusions and Recommendations	18
5.1 Group Studies.....	18
5.2 Embedded Extreme Scale Study	19
6. References.....	21
List of Acronyms	22
Appendix A: Exascale Computing Study Report.....	23
Appendix B: Exascale Computing Software Study Report.....	321
Appendix C: Exascale Computing Resiliency Study Report	481
Appendix D: Embedded Extreme Scale systems Study Report	511

LIST OF TABLES

<u>Table</u>		<u>Page</u>
Table 1	Research Recommendations for Addressing the Major Extreme Scale Technology Challenges.....	3
Table 2	Exascale Computing Study Core Committee.....	7
Table 3	Exascale Computing Software Study Core Committee.....	8
Table 4	Exascale Computing Resiliency Study Core Committee.....	8
Table 5	Research Recommendations for Addressing the Major Extreme Scale Technology Challenges.....	18

PREFACE

This document is the draft Final Technical Report for Georgia Institute of Technology (Georgia Tech, GT) Project 210667V, “Exascale Computing Study” (ECS). The ECS was sponsored by the U.S. Defense Advanced Research Projects Agency (DARPA), contracting through the U.S. Air Force Research laboratory (AFRL), and was conducted over the period May 2007 through December 2010.

The ECS was comprised of four major sub-studies:

- ◆ The original Exascale Computing Study, sometimes called the “hardware study” [1];
- ◆ The Exascale Computing Software Study (ECSS) [2] ;
- ◆ The Exascale Computing Resiliency Study (ECSR); and
- ◆ The Embedded Extreme Scale (EES) systems study [3].

The results and recommendations of these studies, particularly the first three, significantly influenced the focus and structure of the subsequent Ubiquitous High Performance Computing (UHPC) program initiated by DARPA in 2010 [4].

Each of these studies has been previously documented by technical reports delivered under this project. The purpose of this report is to summarize the results and recommendations of the project, and thus of these four studies, as a whole. This report also gathers those four reports into a single document for the convenience of the extreme scale high performance computing community. They appear in their entirety as appendices A through D of this report.

Following are lists of contributors to each of the four ECS study reports, with their affiliations. Georgia Tech (GT) is grateful to these individuals and their institutions for their contributions to the exascale computing studies.

Exascale Computing Study Report Contributors:

Keren Bergman Columbia University	Shekhar Borkar Intel
Dan Campbell Georgia Tech Research Institute	William Carlson Institute for Defense Analyses
William Dally Stanford University	Monty Denneau IBM T. J. Watson Research Laboratories
Paul Franzon North Carolina State University	William Harrod DARPA
Kerry Hill Air Force Research Laboratory	Jon Hiller Science & Technology Associates
Sherman Karp Consultant	Stephen Keckler University of Texas
Dean Klein Micron Technology	Peter Kogge University of Notre Dame

(continued next page)

PREFACE (continued)

Exascale Computing Study Report Contributors (concluded):

Robert Lucas
University of Southern California
Information Sciences Institute
Al Scarpelli
Air Force Research Laboratory
Allan Snively
University of California San Diego
R. Stanley Williams
Hewlett-Packard Laboratories

Mark Richards
Georgia Institute of Technology
Steven Scott
Cray
Thomas Sterling
Louisiana State University
Katherine Yelick
University of California Berkeley

Exascale Computing Software Study Report Contributors:

Saman Amarasinghe
Massachusetts Institute of Technology
William Carlson
Institute for Defense Analyses
William Dally
Stanford University
Mary Hall
University of Utah
William Harrod
DARPA
Jon Hiller
Science & Technology Associates
Charles Koelbel
Rice University
Peter Kogge
University of Notre Dame
Daniel Reed
Microsoft
Robert Schreiber
Hewlett Packard Laboratories
Al Scarpelli
Air Force Research Laboratory
Allan Snively
University of California San Diego

Dan Campbell
Georgia Tech Research Institute
Andrew Chien
Intel
Elmootazbellah Elnozahy
IBM
Robert Harrison
Oak Ridge National Laboratory
Kerry Hill
Air Force Research Laboratory
Sherman Karp
Consultant
David Koester
MITRE
John Levesque
Oak Ridge National Laboratory
Vivek Sarkar
Rice University
Mark Richards
Georgia Institute of Technology
John Shalf
Lawrence Berkeley National Laboratory
Thomas Sterling
Louisiana State University

PREFACE (concluded)

Exascale Computing Resiliency Study Report Contributors:

Ricardo Bianchini,
Rutgers University
Elmootazbellah Elnozahy,
IBM
Forest Godfrey,
Cray
Adolfy Hoisie,
Los Alamos National Laboratory
Rami Melhem,
University of Pittsburgh
Partha Ranganathan,
Hewlett Packard Laboratories

Tarek El-Ghazawi,
George Washington University,
Armando Fox,
University of California, Berkeley
William Harrod,
DARPA
Kathryn McKinley,
University of Texas
James Plank,
University of Tennessee
Josh Simons,
Sun Microsystems

Embedded Extreme Scale Systems Study Report Contributors:

Daniel P. Campbell
Georgia Tech Research Institute
Jason Poovey
Georgia Institute of Technology

Thomas Conte
Georgia Institute of Technology
Mark A. Richards
Georgia Institute of Technology

Timothy Scott
Georgia Institute of Technology

ACKNOWLEDGEMENTS

This project was sponsored by the Information Processing Technology Office (IPTO) of the US Defense Advanced Research Projects Agency (DARPA) and was conducted under the administration of the U.S. Air Force Research Laboratory under contract FA8650-07-C-7724. The authors would like to thank Dr. William Harrod of DARPA/IPTO, the Exascale Computing Study program manager, and Ms. Kerry Hill and Mr. Al Scarpelli of AFRL for their support of this effort. Special thanks are due to Mr. Jon Hiller of Science and Technology Associates and to Dr. Sherman Karp for their guidance and support of the various studies conducted under this project.

Numerous members of the high performance computing community contributed to the constituent studies that comprise this project. Those that were core members of the study groups and who directly aided in authoring the reports were named in the Preface of this document. Thanks are also due, however, to numerous additional persons who participated in individual meetings as invited guest subject matter experts. While too many to enumerate here, these persons are identified in the appendices of the Exascale Computing Study report (Appendix A of this report) and the Exascale Computing Software Study report (Appendix B).

1. SUMMARY

1.1 Goals of the Project

The primary goal of this project was to determine the technical developments needed to enable a 1,000× increase in the computational capabilities of high performance computing (HPC) systems by the 2015 time frame at scales ranging from chassis-level embedded systems to data center systems. Such systems are called *extreme scale* computing systems. To the extent current technology trends were deemed incapable of supporting such increases, the project was charged with identifying the major obstacles and targeted research investments to overcome them. A second goal of the project was to develop design, analysis, and implementation techniques applicable to embedded extreme scale (EES) computing systems.

1.2 Project Approach

To address these goals, Georgia Tech (GT) organized and conducted an Exascale Computing Study (ECS) and an Exascale Computing Software Study (ECSS) on behalf of DARPA. The ECS addressed hardware and system architecture issues in the development of exascale computing systems. The ECSS addressed software technologies, approaches, and methodologies that must be in place to effectively utilize the extreme levels of concurrency expected in exascale computing technology. A third Exascale Computing Resiliency Study (ECRS), conducted by DARPA but facilitated by this project, identified the key technologies, approaches, and methodologies that must be in place to provide practical fault management in exascale systems. Brief summaries of the methods, results, and recommendations of each of these three studies are given in sections 3 through 5 of this report. Full details of each are in their respective final reports, which are included in their entirety as Appendices A, B, and C of this report.

The research on EES systems was conducted entirely by GT. This portion of the project was divided into five sub-tasks, each conducting initial work on an area of interest to the development of extreme scale systems: an internal mini-study of design environment requirements for terascale embedded computing; characterization of embedded computing applications projected to require terascale computing capability in the 2015 time frame; investigation of new concepts for inherently low-power computational algorithms; continued development of high performance computational libraries for advanced graphics processing units (GPUs); and initial consideration of programmability metrics for extreme scale systems. Some of these areas are also applicable to all size classes of extreme scale systems.

1.3 Principal Results and Conclusions

Taken as a whole, the ECS, ECSS, and ECRS resulted in a clear identification of the technology challenges in achieving extreme scale computing capability by the 2015 time frame that would not be met by current commercial technology development trends. At the highest level, these are the power and energy challenge; the concurrency challenge; the resiliency challenge; and the memory and storage challenge. Because these studies were not chartered to focus on device technology, they concentrated on the first three challenges. Both the original ECS and the ECSS

showed that the concurrency challenge could be usefully separated into considerations of massive parallelism, referring to the sheer number of threads required to achieve extreme scale performance, and locality, referring to the dominant effect of data access and movement on power consumption. The resiliency study made it clear that the conventional practice of checkpoint and restart is a non-starter in extreme scale systems, especially at the data center scale. Thus, entirely new techniques are needed for fault management.

The studies also made clear that the challenges are not orthogonal. For instance, the energy/power challenge must be addressed at multiple levels, from device technology through architecture, systems software, and application software. Management of locality in the latter two directly impacts power and energy consumption. As another example, circuit techniques likely to be used for power reduction will also likely *increase* the frequency of device errors, so corresponding attention to reliable circuit design will be needed to compensate.

The EES study task initiated research on several fronts related to practical extreme scale system development, especially at the smaller embedded scale. The design environment mini-study suggested renewed emphasis on hierarchical, statistical simulation as a pragmatic design methodology for predictable systems. The applications analysis task provided the initial development of streaming sensor challenge problems (SSCPs) that establish the computational requirements of near-future streaming sensor applications for the Department of Defense. With further development, the SSCP will also provide an abstracted, publicly releasable sample application that will aid in EES system design and development. Continued development of the GPU Vector, Signal, Image Processing Library (GPU VSIPL) enhances one valuable tool for productive high performance software development for EES systems. The algorithm memory use analysis laid the groundwork for additional research in designing algorithms for minimum power while maintaining high performance. Finally, the early small-scale programmability metric proposal and experiments provided a basis for more extensive research into practical metrics for extreme scale programmability metrics.

1.4 Major Recommendations

The ECS, ECSS, and ECRS studies identified several key challenges that must be addressed if extreme scale systems are to be achieved by the 2015 time frame. Table 1, adapted from Dally [5], compactly encapsulates the most important of the general areas of research recommended by the extreme scale studies for addressing some of the major challenge areas. For each challenge area, research opportunities are separated into those primarily associated with the system hardware, computer architecture, programming and software, and applications. This table outlines a comprehensive program of research investment, ranging from the circuit level to the end-user application and algorithm design. While some of these areas may be more critical than others, it is expected that progress will be required in all of them before viable extreme scale systems can be achieved. More detailed recommendations are given in each of the group study reports in Appendices A through C.

A particular finding of the studies taken as a whole was the need for a new *execution model* for extreme scale systems. An execution model is paradigm for organizing and carrying out computation across all levels of the computer system from programming models and languages, through compilers and runtime systems, to operating systems, to system and

microarchitectures. As such, it provides the conceptual scaffolding for codesign of each of these system elements.

<i>Table 1. Research Recommendations for Addressing the Major Extreme Scale Technology Challenges</i>				
System Level	Power Efficiency	Massive Parallelism	Locality	Resiliency
Applications	Locality-optimized applications	Combined strong, weak, and “new-era” weak scaling	Locality-optimized applications	Application-level checking
Programming Systems	Locality expression and optimization	Parallelism expression and extraction	Locality expression and optimization	Redundancy generation, ALC support
Architecture	Low-overhead processors, power adaptive	Efficient communication & synchronization mechanisms	Agile memory systems	Checkpoint restart, ECC, self-checking
Hardware	Optimized threshold and supply voltages, energy-efficient comm. circuits			Reliable circuits

Source: Adapted from [5].

No further work is planned at this time on design methodology for EES systems, but it is recommended that the other four elements of the EES study be continued. The SSCP is currently planned to be refined and expanded under the UHPC program to include multiple disparate sensors and a sensor fusion stage. In addition, it will be “productized” to a user-ready artifact, with a formal specification, reference serial code, input data or generators, benchmarking rules, and so forth. The proposed programmability metric requires significant research to determine if it can serve as a practical basis for a programmability metric for large scale UHPC and EES systems. Additional work is needed in the concept development; programming problems; and scoring methodology. This work will also continue as part of the UHPC program.

It is recommended that the work on algorithmic data motion analysis be continued under UHPC also. Much work remains to build from the initial demonstrations to a practically useful methodology. The correlation between data motion and actual energy costs must be established, as must the applicability of the proposed methods to large scale problems. Also, the analysis of locality and data motion, and the prediction of algorithm energy efficiency based on these analyses, is not an end in itself, but the basis for developing design techniques that lead to high performance numerical algorithms with inherently low data motion and energy costs.

Finally, it is recommended that GT continue development of the GPU VSIPL library as a useful productivity tool for EES systems, and possibly larger extreme scale systems as well. Immediate needs are concentrated on debugging and validation of various linear system solvers, and on updating of the VSIPL Test Suite.

2. INTRODUCTION

2.1 Goal of the Project

The primary goal of this project, conducted from mid-2007 to late 2010, was to determine the technical developments required to enable a 1,000× increase in the computational capabilities of HPC systems by the 2015 time frame, relative to current high performance computer systems. To the extent current technology trends were deemed incapable of supporting such increases, then the project was also charged with identifying the major obstacles, and targeted research investments needed to overcome them. A secondary goal of the project was to develop design, analysis, and implementation techniques specifically applicable to advanced embedded HPC systems.

2.2 Extreme Scale Computing Systems

Initially, the goal was described as *exascale* computing in recognition of the fact that the first computer to achieve a speed exceeding one petaflops/s (Pflops), as rated by the “Top 500” list, was achieved in June 2008 [7]. The use of the word “exascale” (instead of exaflops) reflected a deliberate focus on looking beyond just floating point operations to a broader set of performance metrics that include integer, network, and memory system performance. The change from exascale to “extreme scale” reflected the project’s focus on not just high-end data center scale supercomputers, but more broadly the ability to perform computations of both traditional and emerging significance to the Department of Defense (DoD) at multiple system scales.

Although most of the studies conducted under this project focused on the high-end, true exascale systems to identify technology barriers, it is expected that the technology needed to enable these true exascale systems would also enable a 1000× increase, relative to current capability, for smaller-scale computers. Specifically, it is assumed that “departmental” systems could be constructed that operate at petascale levels in a form factor of one or two standard cabinets, and that “embedded” systems could be constructed that operate at terascale levels in a VME chassis physical scale. The moniker *extreme scale systems* was adopted to refer collectively to this full range of computing capabilities and physical sizes.

2.3 DARPA’s Extreme Scale Computing Studies

In 2007-2008, Georgia Tech (GT) organized and conducted under this project an Exascale Computing Study on behalf of DARPA. The ECS addressed hardware and system architecture issues in the development of exascale computing systems, focusing particularly on power and energy, parallelism, and concurrency. The final report of the ECS has previously been publicly released [1], and is included as Appendix A of this report.

Among other issues (to be discussed in Section 4), the ECS identified the development of effective parallel programming methodologies for systems having extreme degrees of concurrency, and exascale system resiliency as critical issues. Consequently, in 2008-2009 GT

organized and conducted a supplemental Exascale Computing Software Study to identify the key software technologies, approaches, and methodologies that must be in place to effectively utilize the extreme levels of concurrency expected in exascale computing technology. The final report of the ECSS has previously been publicly released [2], and is included in its entirety as Appendix B of this report.

In 2008-2009, DARPA also conducted an Exascale Computing Resiliency Study, facilitated by this project. The goal of the ECRS was to identify the key technologies, approaches, and methodologies that must be in place to ensure scalability of mean time to failure (MTTF) in exascale systems. The “white paper” summarizing the ECRS study is included in its entirety as Appendix C of this report.

2.4 Embedded Extreme Scale Study

In 2008, GT undertook a study several of key issues in realizing embedded extreme scale computing systems. The EES study comprised five principal sub-tasks. The first focused on identifying embedded computing applications projected to require terascale computing capability in the 2015 time frame. GT analyzed the computational, communication, memory, and other requirements of the selected applications. Key functional kernels and application benchmarks representative of the selected applications were identified. The second sub-task focused on a design environment for terascale embedded computing. GT assessed current performance modeling and prediction methods applicable to embedded computing systems.

The third sub-task considered new concepts for inherently low-power computational algorithms. Under this part of the effort, GT investigated techniques for developing new algorithms and improvements to existing algorithms that jointly optimize the energy and runtime required for computation of selected functional kernels. Under the fourth sub-task, GT conducted related work in the development of high performance computational libraries for advanced graphics processing units (GPUs). Finally, the fifth sub-task addressed the problem of designing effective programmability metrics for extreme-scale systems.

The results of this task were documented in a technical report that is included in its entirety as Appendix D of this report.

2.5 Organization of This Report

The bulk of the work conducted under this project is described in complete detail in the four previous reports included as Appendices A – D of this report. These reports describe the participants, procedures, results, and recommendations for their respective areas of focus. Consequently, the body of this report is limited to a high-level description of the common aspects of the studies and the most important overall conclusions of the project as a whole.

Section 3 of this report describes the common methods and procedures used across the studies. Section 4 summarizes the most important conclusions of each, and the conclusions of the project as a whole. Similarly, Section 5 summarizes the most important recommendations of each study, and of the project as a whole. Appendices A, B, C, and D are respectively the final products of the ECS, ECSS, ECRS, and EES studies.

3. METHODS AND PROCEDURES

3.1 Group Studies

The ECS, ECSS, and ECRS were all similar in their organization and conduct. For the ECS and ECSS, DARPA and GT began by identifying a community leader to serve as the technical chair of the study. These were Prof. Peter Kogge of Notre Dame for the ECS, and Prof. Vivek Sarkar of Rice University for the ECSS. GT, DARPA, and the chair then invited a number of appropriate subject matter experts (SMEs) from the HPC community to serve as the core committee for each study. Tables 2 and 3 detail the core committee members, including government participants, for the ECS and ECSS. The ECRS committee was organized directly by DARPA with Dr. Elmootazbellah Elnozahy of IBM as its chair, and study committee members selected by DARPA and Dr. Elnozahy. Table 4 lists the core committee members for the ECRS.

<i>Table 2. Exascale Computing Study Core Committee</i>	
Keren Bergman, Columbia University	Shekhar Borkar, Intel
Dan Campbell, Georgia Tech Research Institute	William Carlson, Institute for Defense Analyses
William Dally, Stanford University	Monty Denneau, IBM T. J. Watson Research Laboratories
Paul Franzon, North Carolina State University	William Harrod, DARPA
Kerry Hill, Air Force Research Laboratory	Jon Hiller, Science & Technology Associates
Sherman Karp Consultant	Stephen Keckler, University of Texas
Dean Klein, Micron Technology	Peter Kogge (technical chair), University of Notre Dame
Robert Lucas, University of Southern California Information Sciences Institute	Mark Richards, Georgia Institute of Technology
Al Scarpelli, Air Force Research Laboratory	Steven Scott, Cray
Allan Snively, University of California, San Diego	Thomas Sterling, Louisiana State University
R. Stanley Williams, Hewlett-Packard Laboratories	Katherine Yelick, University of California, Berkeley

<i>Table 3. Exascale Computing Software Study Core Committee</i>	
Saman Amarasinghe, Massachusetts Institute of Technology	Dan Campbell, Georgia Tech Research Institute
William Carlson, Institute for Defense Analyses	Andrew Chien, Intel
William Dally, Stanford University	Elmootazbellah Elnozahy, IBM
Mary Hall, University of Utah	Robert Harrison, Oak Ridge National Laboratory
William Harrod, DARPA	Kerry Hill, Air Force Research Laboratory
Jon Hiller, Science & Technology Associates	Sherman Karp Consultant
Charles Koelbel, Rice University	David Koester, MITRE
Peter Kogge, University of Notre Dame	John Levesque, Oak Ridge National Laboratory
Daniel Reed, Microsoft	Vivek Sarkar (technical chair), Rice University
Robert Schreiber, Hewlett Packard Laboratories	Mark Richards, Georgia Institute of Technology
Al Scarpelli, Air Force Research Laboratory	John Shalf, Lawrence Berkeley National Laboratory
Allan Snively, University of California, San Diego	Thomas Sterling, Louisiana State University

<i>Table 4. Exascale Computing Resiliency Study Core Committee</i>	
Ricardo Bianchini, Rutgers University	Tarek El-Ghazawi, George Washington University,
Elmootazbellah Elnozahy (technical chair), IBM	Armando Fox, University of California, Berkeley
Forest Godfrey, Cray	William Harrod, DARPA
Adolfy Hoisie, Los Alamos National Laboratory	Kathryn McKinley, University of Texas
Rami Melhem, University of Pittsburgh	James Plank, University of Tennessee
Partha Ranganathan, Hewlett-Packard Laboratories	Josh Simons, Sun Microsystems

Each group conducted a series of working meetings attended by all available members of the core study committee. The technical focus of the meetings was determined by the technical chair in consultation with DARPA and GT. Details of the meetings held are included in the individual study technical reports in the appendices to this report. For most meetings, a number of additional SMEs, appropriate to the subject of that meeting, were invited to attend and participate. Some meetings, typically the first and last, were limited in whole or part to core members to focus on study planning or development of the final report.

In addition to participating technically, GT served as organizer and facilitator of the ECS and ECSS study meetings. GT assisted in site selection, including hosting some meetings at GT's facilities. GT also arranged for local meeting support as required, including local travel information, audio/visual equipment as required, internet access for participants, and refreshments. In addition, GT provided travel funding and honoraria where required to enable participation by SMEs. The technical chairs and certain other participants were supported under subcontract from GT.

3.2 Embedded Extreme Scale Study

Unlike those described in the preceding section, the Embedded Extreme Scale (EES) study task was conducted entirely by GT and did not involve community-wide group meetings. As actually implemented, the EES study included five tasks:

1. Analysis of design environment requirements for terascale embedded computing.
2. Analysis of embedded computing applications projected to require terascale computing capability in the 2015 time frame.
3. Investigating new concepts for inherently low-power computational algorithms.
4. Continued development of high performance computational libraries for advanced graphics processing units (GPUs).
5. Initial consideration of selected metrics for extreme scale systems.

For the first task, GT conducted an internal mini-study group in the summer of 2009 to address issues in embedded terascale system design. A series of biweekly meetings and discussions involving a group of about one half-dozen GT faculty were held. The discussions addressed both hardware and software issues in designing systems of this class, ultimately focusing on the problem of "predictable design".

The second task was addressed through the identification and computational analysis of three major embedded real-time applications: ground moving target indication (GMTI) by radar; persistent wide area surveillance (WAS) by radar; and autonomous ground vehicles. For each of these applications, GT analyzed the computational, communication, memory, and other requirements of the selected applications, considering several variations requiring different levels of computing capability, from present-day technology through future extreme scale technology. In the case of the streaming imaging sensor, a "streaming sensor challenge

problem” benchmark was defined to provide an unrestricted, abstracted, and scalable representation of this class of computing problem.

The third task, concepts for low-power numerical algorithms, was limited to preliminary analysis of metrics for data motion and the development of an initial MATLAB-based infrastructure for measuring data motion in simple, canonical algorithms such as the fast Fourier transform (FFT), finite impulse response (FIR) digital filters, and matrix multiplication. This approach was based on the assumption, consistent with the ECS study, that data motion through the memory hierarchy is one of the major energy consumers in extreme scale computing, and thus one of the major obstacles to successful implementation of such machines.

The fourth task continued development of an implementation of the Vector, Signal, Image Processing Library (VSIPL) [8] application programming interface (API) for GPUs. The resulting library is known as GPU VSIPL [9]. This effort addressed extension of the library to include important matrix decompositions and linear equation solvers from the VSIPL “Core profile”, and update of the VSIPL Test Suite.

The fifth task considered metrics for extreme scale systems. While traditional metrics such as energy efficiency, performance in operations per second on appropriate workloads, and scalability remain important, the extreme concurrency required to utilize extreme scale architectures is expected to greatly increase programming difficulties for end users. Consequently, metrics for such characteristics as programmability and productivity that are applicable to these systems are needed. It is expected that new metrics for resiliency and dependability and security will also be required. Under this project, a limited initial study of programmability metrics issues was initiated. After an initial literature review, GT proposed a programmability metric methodology and tested it on a very small scale as a means of identifying issues and refining the proposed metric.

4. RESULTS AND DISCUSSION

The complete results of each of the major studies under this project (ECS, ECSS, ECRS, and EES study) are contained in their respective reports in Appendices A through D. In this section, a very brief summary of the major results is given.

4.1 Group Studies

The original exascale computing study concluded that there are four major challenges to achieving exascale systems where current technology trends are simply insufficient, and significant new research is needed:

- ◆ *The Energy and Power Challenge* is the most pervasive of the four. The ECS study group was unable to project any combination of currently mature technologies that will deliver systems with sufficient performance in any size class at the desired power levels. A key observation of the study was that it may be easier to solve the power problem associated with base computation than it will be to solve the problem of transporting data from one site to another, whether on the same chip, between closely coupled chips in a common package, between different racks on opposite sides of a large machine room, or in storing data in the aggregate memory hierarchy.
- ◆ *The Memory and Storage Challenge* recognizes the lack of currently available technology to retain data at high enough capacities, and access it at high enough rates, to support the desired application suites at the desired computational rate, and still fit within an acceptable power envelope. This challenge applies to local memory and storage (multi-level caches), main memory (dynamic random access memory [DRAM] today) and secondary storage (rotating disks today).
- ◆ *The Concurrency and Locality Challenge* arises from the flattening of silicon clock rates and the end of increasing single thread performance, leaving explicit, largely programmer-visible parallelism as the only mechanism in silicon to increase overall system performance. While this affects all three classes of systems (data center, departmental, and embedded), projections for the data center class systems in particular indicate that applications may have to support upwards of a billion separate threads to efficiently use the hardware.
- ◆ *A Resiliency Challenge* that deals with the ability of a system to continue operation in the presence of either faults or performance fluctuations. This concern grew out of not only the explosive growth in component count for the larger classes of systems, but also out of the need to use advanced technology, at lower voltage levels, where individual devices and circuits become increasingly sensitive to local operating environments, and new classes of aging effects become significant.

While the latter three challenges grew out of consideration of the high end systems, they are certainly not limited to that class. The explosive growth of highly multi-core microprocessors and their voracious appetite for more random access memory (RAM) creates smaller-scale versions of these same challenges for the smaller extreme scale systems.

Thus, the ECS study established the assumption that all three classes of extreme scale systems will be built using massive multi-core processors with hundreds of cores per chip; that their performance will be driven by parallelism and constrained by energy; and that they will be subject to frequent faults and failures. Beginning from the ECS results, the exascale computing software study focused on the implications of the concurrency and energy challenges for system and application software, tools, and APIs. Also included was identification of opportunities for software-hardware co-design, as well interfaces between applications and system software and between system software and hardware. Extreme scale algorithms and application software were not included in the ECSS scope.

The concurrency challenge will require extreme scale software to expose at least 1000× more concurrency in applications for extreme scale systems, relative to current systems. It is further exacerbated by the projected memory-computation imbalances in extreme scale systems, with bytes/ops ratios that may drop to values as low as 10^{-2} , where “bytes” and “ops” represent the main memory and computation throughput capacities of the system, respectively. These ratios will result in 100× reductions in memory per core relative to petascale systems, with accompanying reductions in memory bandwidth per core. Thus, a significant fraction of software concurrency in extreme scale systems must come from exploiting more parallelism within the computation performed on a single datum, i.e., from strong scaling or from the “new-era” weak scaling discussed in Chapter 4 of the ECSS report (Appendix B). Strong scaling often involves more frequent communication and synchronization than weak scaling, which in turn contributes to the energy efficiency challenge since data movement and synchronization are major contributors to energy costs. Another major obstacle to achieving a large degree of concurrency arises from the serialization bottlenecks in current system software approaches to communication and synchronization.

All three classes of Extreme Scale systems will be expected to deliver their 1000× improvements in computation capability while essentially remaining within the power budgets of current systems. An aggressive hardware design for data center-sized systems will need at least 60MW of power to achieve an exa-op level of performance, under highly idealized zero-overhead assumptions for software. This energy challenge affects system software design because when current software overheads are taken into account, it is clear that extreme scale capability cannot be achieved without a significant redesign of the system software stack.

The ECSS found that a 1000× increase in computation capability for each class of extreme scale system will only be achievable through radical re-design of the underlying execution model and system software and hardware. Current execution models and system designs will not work at extreme scale because of their sequential foundations and inherent energy inefficiencies. In addition, use of current execution models at extreme scale will result in prohibitively large costs in programmability. While the High Productivity Computing Systems (HPCS) [6] program and other recent efforts have demonstrated reductions in the human effort required to develop high-productivity software for current petascale systems, they do not address the energy-constrained many-core parallelism and heterogeneous processors expected in extreme scale architectures. Also, while there is some overlap between system software requirements for extreme scale and those for large scale commercial data centers, there are also significant differences. Commercial system software for cloud computing is primarily focused on optimizing throughput capacity of

independent jobs, whereas system software for extreme scale computing must be capable of delivering a 1000× increase in parallelism to a single job.

Also building on the challenges identified by the original ECS, the extreme scale resiliency study considered the resiliency challenge. It found that today, 20% or more of the computing capacity in a large high-performance computing system is wasted due to failures and recoveries. Typical mean time between failures (MTBF) is from 8 hours to 15 days. As systems increase in size to field petascale computing capability and beyond, the MTBF will go lower and more capacity will be lost. Indeed, it is not difficult to project exascale systems that will use 100% of their time in checkpointing and recovery using current protocols, thus performing no useful work at all at immense costs in dollars and energy.

The ECRS showed that the programming model based on flat message passing using the message passing interface (MPI) is central to the current problem in providing reliability. This model does not offer any failure containment, and thus a failure in one node in the system triggers a whole-system failure. As systems continue to increase in size, this approach is not tenable. The study also analyzed current trends in technology and showed that power management, the recent trend toward heterogeneous computing, and the expected increase in system size all will interact negatively with resilience at the system level. To counter these trends, the ECRS considered an extensive palette of resilience techniques at all levels of the system. At the hardware level, stable semiconductor memory devices and monitoring of hardware threads can be applied. At the system runtime level, power management methods can be combined with efficient state capture/recovery and virtualization. Particularly novel is the suggestion to combine resilience-oriented compiler and programming methods with statistical machine learning methods for fault detection, isolation, and recovery to create a form of “self-aware” system.

4.2 Embedded Extreme Scale Study

4.2.1 Design Environments for Terascale Embedded Computing

The internal study group considering design environments for terascale embedded computing began with a brief review of common embedded computing system design approaches such as incremental design, analytical approximation, and simulation-based design. The study concentrated on hierarchical simulation-based design as the most practical for extreme scale systems, and in turn on the particular issue of *predictable design*, that is, the property wherein a system’s performance is predictable to within some certain acceptable error margin, before the construction of the system. This property is not a given due to the multitude of complex mechanisms employed to accelerate computer architecture performance, such as the use of caches and branch prediction, as well as uncertainties in the simulations themselves. Furthermore, the greater the desired performance from a given architecture, the more techniques that must be deployed to achieve that performance, and the greater the uncertainty and the variability in the predicted performance. Another source of difficulty is simulation loads. Attempts to improve simulation accuracy generally involve more complete and detailed simulations, rapidly reducing the amount of system time that can be simulated effectively. The study then considered the use of statistical sampling methods in simulation to improve the tradeoff between accuracy and simulation load.

4.2.2 Extreme Scale Embedded Computing Applications

The analysis of extreme scale embedded computing applications considered two technologies associated with wide area surveillance using radar systems from airborne platforms. The first was an advanced “knowledge-based” form of space-time adaptive processing (STAP) using primarily linear filters and linear algebraic computations. The computational requirements of this system were considered for five different use scenarios, resulting in computational loads of 167 gigaflops/s (Gflops) to 572 Pflops (0.572 exaflops/s [Eflops]). In today’s technology, the low-end system could be implemented on a single GPU, but the high end exceeds the capacity of the current fastest supercomputer in the world (at this writing, the Chinese Tianhe-1A computer, at 2.57 Pflops [10]) by two orders of magnitude. The other WAS technology was a “video synthetic aperture radar” (SAR) capability capable of generating SAR images of an area at 1-second update rates using primarily FFTs and backprojection calculations. Again, multiple scenarios were considered, ranging from images on the order of 1 megapixel in size, to images over 4,000 megapixels in size. The resulting computational loads ranged from 43.8 Gflops to 12.1 Pflops.

The video SAR system was then used as the basis for defining a “streaming sensor challenge problem” (SSCP) which married the image formation with an image analysis stage that applied coherent change detection and constant-false alarm rate processing to extract targets from the SAR video stream. This algorithm, which combined FFTs and backprojection with matrix multiplies and correlations, was analyzed for four different use scenarios, resulting in estimated loading from 440 Gflops to 185 Pflops. The SSCP became the basis for the first version of the UHPC streaming sensor challenge problem [4].

GT also analyzed the computational requirements of the software suite used in GT’s “Sting” autonomous vehicle, which participated in the 2007 DARPA “Urban Challenge” event. Essentially a computer vision and robotics problem, this application represents a very different computational architecture. As implemented, the computational load for the Sting software was estimated to be very moderate, on the order of 550 Mflops, too low to be of interest for extreme scale systems. GT then postulated an “extreme scale autonomous vehicle” that substituted more advanced and complex algorithms for key operations. Estimates of computational loads were as high as 16.2 Pflops, depending on system parameters.

4.2.3 Low-Power Computational Algorithms

Understanding how an application accesses its data from memory is important in understanding its performance, as accessing successively higher levels of a memory hierarchy comes at an exponentially increasing cost of energy and time. Often, memory accesses of known algorithms occur in a pattern that can be predicted and modeled. By studying these patterns, insight can be gained into how to best minimize energy costs in applications using these algorithms.

For this task, GT began a study of the data movement and memory access properties of key numerical algorithms on modern architectures, with the eventual goal of developing methods to design and optimize algorithms to significantly reduce their energy cost at little to no performance cost, compared to current best-practice algorithms for the same functions. The

algorithms considered were two variations each of finite impulse response (FIR) filters, fast Fourier transforms, and matrix multiplication. GT developed a simple method for semi-automatically generating a trace of the data memory access patterns of these algorithms using MATLAB tools, and analyzing that data to obtain information about the application's spatial and temporal locality and "data motion". The locality metrics are measures of the efficiency (or lack thereof) of an algorithm in minimizing the number of memory accesses required, while the data motion metric is a combination of spatial and temporal locality that also takes into account variations in access energy at different levels of a typical memory hierarchy. These tools were used to calculate quantitative metrics that can be used for making comparison of these different applications. Preliminary experiments confirmed that different algorithms for computing the same function can differ significantly in measures of locality and data motion.

4.2.4 High Performance Libraries for Advanced Graphics Processing Units

During the course of this project, the use of graphics processing units (GPUs) for "general purpose" scientific computing, including signal processing, has continued to grow rapidly in interest and acceptance in the HPC community across all scales, from embedded to data center. GPUs achieve high computational throughput by many of the same means projected by the ECS study for future large scale systems, and thus present a useful proxy for extreme scale systems. VSIPL is a portable API for implementing high-performance signal processing applications while retaining platform independence. It supports memory abstractions for utilizing coprocessors with disjoint memory spaces. Intermediate results are not transferred between system and GPU memory, avoiding unnecessary latencies and communications overhead and distinguishing VSIPL from other signal processing libraries that permit random access to data.

GT's GPU VSIPL library includes most of the VSIPL "Core profile" functionality, with the exception of some of the linear equation solvers and random number functions, but with the addition of a large number of matrix arithmetic operations not included in the VSIPL Core profile. Under the EES study portion of this project, GT continued enhancing GPU VSIPL to include important matrix decompositions and linear equation solvers from the Core profile. GT also updated the VSIPL Test Suite [8]. Details are given in Appendix D.

4.2.5 Metrics for Extreme Scale Systems

UHPC systems, which include EES systems at the embedded scale, will be characterized and evaluated by a number of metrics. The UHPC program is considering the development of metrics to characterize the following aspects of UHPC systems:

- | | |
|---------------------|-----------------|
| ◆ Performance | ◆ Scalability |
| ◆ Energy efficiency | ◆ Dependability |
| ◆ Resiliency | ◆ Security |
| ◆ Self-awareness | ◆ Productivity |

Metrics for some of these aspects, such as performance or scalability (weak and strong), are well known. Other aspects are harder to characterize. There are no agreed-upon metrics for resiliency or self-awareness, for instance. While the HPCS program addressed productivity, it

has not to date resulted in widely-accepted productivity metrics. In the UHPC program, it is expected that in the near term, the emphasis will be on programmability rather than a broader definition of productivity.

A system is considered highly programmable if it does not require application programmers to explicitly manage system complexity in order to achieve their performance and time to solution goals. As part of the EES study task on this project, GT investigated techniques for assessing the programmability of high performance parallel systems. Based on this investigation, an initial proposal for a methodology to evaluate programmability was developed based on the concepts of “parallel patterns” and “cognitive dimensions” from the software and human-computer interface communities. In developing the proposed programmability metric, GT conducted two small experiments to help define and demonstrate this initial methodology on the Quicksort and MapReduce algorithms, comparing Pthreads and OpenMP implementations on a small (laptop) machine. Details of the proposed metric and the experimental results are given in the EES study report in Appendix D.

4.3 Discussion

Taken as a whole, the three group studies conducted under or facilitated by this project resulted in a clear identification of the technology challenges in achieving extreme scale computing capability by the 2015 time frame that would not be met by current commercial technology development trends. At the highest level, these are the power and energy challenge; the concurrency challenge; the resiliency challenge; and the memory and storage challenge. Because these studies were not chartered to focus on device technology, they focused on the first three challenges. Both the original ECS and the ECSS showed that the concurrency challenge could be usefully separated into considerations of massive parallelism, referring to the sheer number of threads required to achieve extreme scale performance, and locality, referring to the dominant effect of data access and movement on power consumption. The resiliency study made it clear that the conventional practice of checkpoint and restart is a non-starter in extreme scale systems, especially at the data center scale. Thus, entirely new techniques are needed for fault management.

The studies also made clear that the challenges are not orthogonal. For instance, the energy/power challenge must be addressed at multiple levels, from device technology through architecture, systems software, and application software. Management of locality in the latter two directly impacts power and energy consumption. As another example, circuit techniques likely to be used for power reduction will also likely *increase* the frequency of device errors, so corresponding attention to reliable circuit design will be needed to compensate.

The EES study task, as a whole, initiated research on several fronts related to practical extreme scale system development, especially at the smaller embedded scale. The design environment mini-study suggested renewed emphasis on hierarchical, statistical simulation as a pragmatic design methodology for predictable systems. The applications analysis task provided the initial development of streaming sensor challenge problems that establish the computational requirements of near-future streaming sensor applications for the DoD. With further development, the SSCP will also provide a publicly releasable sample application that will aid in EES system design and development. Continued development of GPU VSIPL enhances one

valuable tool for productive high performance software development for EES systems. The research in algorithm memory use analysis laid the groundwork for additional research in designing algorithms for minimum power while maintaining high performance. Finally, the early small-scale programmability metric proposal and experiments provides a basis for more extensive research into practical metrics for extreme scale programmability metrics.

5. CONCLUSIONS AND RECOMMENDATIONS

5.1 Group Studies

The ECS, ECSS, and ECRS studies were chartered to determine whether a 1000× increase in HPC capability, at physical scales from chassis-level embedded systems to data centers, could be achieved by the 2015 time frame and, if not, what obstacles existed and how those obstacles might be addressed. The studies clearly demonstrated that existing technology trends would not achieve the desired capability, and identified several key challenges that must be addressed to remedy this situation.

Table 1, repeated here as Table 5 for convenience, compactly encapsulates the most important of the general areas of research recommended by the extreme scale studies for addressing some of the major challenge areas. For each challenge area, research opportunities are separated into those primarily associated with the system hardware, computer architecture, programming and software, and applications. This table outlines a comprehensive program of research investment, ranging from the circuit level to the end-user application and algorithm design. While some of these areas may be more critical than others, it is expected that progress will be required in all of them before viable extreme scale systems can be achieved.

<i>Table 5. Research Recommendations for Addressing the Major Extreme Scale Technology Challenges</i>				
System Level	Power Efficiency	Massive Parallelism	Locality	Resiliency
Applications	Locality-optimized applications	Combined strong, weak, and “new-era” weak scaling	Locality-optimized applications	Application-level checking
Programming Systems	Locality expression and optimization	Parallelism expression and extraction	Locality expression and optimization	Redundancy generation, ALC support
Architecture	Low-overhead processors, power adaptive	Efficient communication & synchronization mechanisms	Agile memory systems	Checkpoint restart, ECC, self-checking
Hardware	Optimized threshold and supply voltages, energy-efficient comm. circuits			Reliable circuits

Source: Adapted from [5].

A particular finding of the studies taken as a whole was the need for a new *execution model* for extreme scale systems. An execution model is paradigm for organizing and carrying out computation across all levels of the computer system stack from programming models and languages through compilers and runtime systems to operating systems and system and micro architectures. It provides the conceptual scaffolding for deriving each of these system elements in the context of and consistent with all of the others. Some examples of previous execution models are the classic Von Neumann serial model, vector parallelism, and communicating shared processes.

More extensive recommendations are given in each of the group study reports in Appendices A through C.

5.2 Embedded Extreme Scale Study

The EES study performed initial work on several areas of continued interest to the development of extreme scale systems, some primarily applicable to embedded class machines, others applicable to all size classes.

The streaming sensor challenge problem is of significant potential utility in the embedded HPC community and also in the UHPC program. As noted, it has already provided the starting point for development of the UHCP SSCP. It is expected that the SSCP will be refined and expanded in its functional scope under the UHPC program to include multiple disparate sensors and thus also a sensor fusion stage. In addition, it will be “productized” to a user-ready artifact, with a formal specification, reference serial code, any necessary input data or generators, benchmarking rules, and so forth.

Much work remains to build from the initial algorithmic data motion analysis demonstrations conducted under this project to a practically useful methodology. It is critical to establish and validate the relationship between locality and data motion scores and actual energy costs of an algorithm on a modern architecture. For example, how does increasing the temporal locality by 20% affect the energy efficiency of a particular application? Another area of exploration is the effect of multithreading and the use multiple cores on a given algorithm’s locality, and whether specific architectures can be mapped to the calculations to result in more accurate and less generic scores. Yet another is scale. The examples given in Appendix D are very small “toy” problems used to establish basic concepts and definitions. Do the proposed methods and metrics scale to large programs? Finally, the analysis of locality and data motion, and the prediction of algorithm energy efficiency based on these analyses, is not an end in itself. Rather, the intent is to provide the basis for developing design techniques that lead to high performance numerical algorithms with inherently low data motion costs. Another avenue of research is in ways to design such algorithms. One possibility would be to develop methods for analysis of mathematical expressions, perhaps in a factored matrix form, of different algorithms for the same function. Another would be to apply the technology of autotuning, used primarily for improving algorithm speed so far, to the minimization of data motion or energy subject to a minimum performance constraint.

Similarly, the initial experimentation with a proposed programmability metric under this project just scratches the surface. The proposed methodology requires significant additional

research to determine if it can serve as a practical basis for a programmability metric for high performance, large scale UHPC and EES systems. Additional work is needed in the concept development; selection and definition of programming problems; scoring methodology; and application to large-scale applications and machines. For instance, what is the degree of correlation between programmability and cognitive dimensions, since the existence of such a correlation is the fundamental assumption of the proposed methodology? Care must be taken to ensure that the basis set of canonical parallel patterns selected adequately represents the range of applications of interest to the UHPC program. Additional work is also needed on automating scoring of the cognitive dimensions from the parallel codes. Furthermore, means to specify and score the codes are needed that will ensure that performance and programmability are targeted jointly. Finally, while GT believes that the concepts proposed appear applicable to large-scale machines, there are undoubtedly a number of issues that will arise as problem and machine sizes are scaled up. It is expected that portions of this research will continue under the UHPC program.

It is also recommended that GT continue development of the GPU VSIPL library as a useful productivity tool for EES systems, and possibly larger extreme scale systems as well. Immediate needs are further debugging and validation of the singular value decomposition functions; completion and validation of the real and complex QRD decomposition and Toeplitz solver functions; and addition to the library of other solvers from the VSIPL Core profile. The VSIPL Test Suite should be updated as required to provide coverage of all new functionality. In addition, the effort begun under this project to extend the test suite coverage to include all of the element-wise functions present in the VSIPL Core profile should be completed.

6. REFERENCES

- [1] "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems", P. Kogge, editor, September 28, 2008. Available at users.ece.gatech.edu/~mrichard/ExascaleComputingStudyReports/ECS_reports.htm.
- [2] "ExaScale Computing Software Study: Software Challenges in Extreme Scale Systems", V. Sarkar, editor, September 14, 2009. Available at users.ece.gatech.edu/~mrichard/ExascaleComputingStudyReports/ECS_reports.htm.
- [3] "Embedded Terascale System Analysis and Design Environment Report", M. A. Richards, *et al*, Georgia Tech technical report, project 210667V, May 2011.
- [4] Ubiquitous High Performance Computing (UHPC) Program solicitation web site, [www.darpa.mil/Our_Work/I2O/Programs/Ubiquitous_High_Performance_Computing_\(UHPC\).aspx](http://www.darpa.mil/Our_Work/I2O/Programs/Ubiquitous_High_Performance_Computing_(UHPC).aspx).
- [5] W. Dally, personal communication.
- [6] DARPA High Productivity Computing Systems web site, [www.darpa.mil/Our_Work/I2O/Programs/High_Productivity_Computing_Systems_\(HPCS\).aspx](http://www.darpa.mil/Our_Work/I2O/Programs/High_Productivity_Computing_Systems_(HPCS).aspx).
- [7] Top 500 list for June 2008. www.top500.org/lists/2008/06.
- [8] Vector, Signal, Image Processing Library (VSIPL) web site. www.vsipl.org.
- [9] GPU VSIPL web site. gpu-vsipl.gtri.gatech.edu/.
- [10] Top 500 list for November 2010. www.top500.org/lists/2010/011.

LIST OF ACRONYMS

<u>Acronym</u>	<u>Definition</u>
AFRL	Air Force Research Laboratory
API	Application Programming Interface
DARPA	Defense Advanced Research Projects Agency
DRAM	Dynamic Random Access Memory
DoD	Department of Defense (U.S.)
ECS	Exascale Computing Study
ECRS	Exascale Computing Resiliency Study
ECSS	Exascale Computing Software Study
Eflops	Exa floating point operations per second (10^{18} flops)
FIR	Finite Impulse Response
FFT	Fast Fourier Transform
Gflops	Giga floating point operations per second (10^9 flops)
GMTI	Ground Moving Target Indication
GPU	Graphics Processing Unit
GT	Georgia Tech
GTRI	Georgia Tech Research Institute
HPC	High Performance Computing
HPCS	High Productivity Computing Systems
Mflops	Mega floating point operations per second (10^6 flops)
MPI	Message Passing Interface
MTBF	Mean Time Between Failures
MTTF	Mean Time To Failure
Pflops	Peta floating point operations per second (10^{15} flops)
RAM	Random Access Memory
SME	Subject Matter Expert
SSCP	Streaming Sensor Challenge Problem
STAP	Space-Time Adaptive Processing
SAR	Synthetic Aperture Radar
Tflops	Tera floating point operations per second (10^{12} flops)
VSIPL	Vector, Signal, Image Processing Library
UHPC	Ubiquitous High Performance Computing
WAS	Wide Area Surveillance

APPENDIX A

FINAL REPORT OF EXASCALE COMPUTING STUDY

ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems

Peter Kogge, Editor & Study Lead

Keren Bergman

Shekhar Borkar

Dan Campbell

William Carlson

William Dally

Monty Denneau

Paul Franzon

William Harrod

Kerry Hill

Jon Hiller

Sherman Karp

Stephen Keckler

Dean Klein

Robert Lucas

Mark Richards

Al Scarpelli

Steven Scott

Allan Snavely

Thomas Sterling

R. Stanley Williams

Katherine Yelick

September 28, 2008



**The views expressed are those of the authors and do not reflect the
official policy or position of the Department of Defense or the U.S. Government.**

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ECS Report

This page intentionally left blank.

ECS Report

DISCLAIMER

The following disclaimer was signed by all members of the Exascale Study Group (listed below):

I agree that the material in this document reflects the collective views, ideas, opinions and findings of the study participants only, and not those of any of the universities, corporations, or other institutions with which they are affiliated. Furthermore, the material in this document does not reflect the official views, ideas, opinions and/or findings of DARPA, the Department of Defense, or of the United States government.

Keren Bergman
Shekhar Borkar
Dan Campbell
William Carlson
William Dally
Monty Denneau
Paul Franzon
William Harrod
Kerry Hill
Jon Hiller
Sherman Karp
Stephen Keckler
Dean Klein
Peter Kogge
Robert Lucas
Mark Richards
Al Scarpelli
Steven Scott
Allan Snively
Thomas Sterling
R. Stanley Williams
Katherine Yelick

This page intentionally left blank.

FOREWORD

This document reflects the thoughts of a group of highly talented individuals from universities, industry, and research labs on what might be the challenges in advancing computing by a thousand-fold by 2015. The work was sponsored by DARPA IPTO with Dr. William Harrod as Program Manager, under AFRL contract #FA8650-07-C-7724. The report itself was drawn from the results of a series of meetings over the second half of 2007, and as such reflects a snapshot in time.

The goal of the study was to assay the state of the art, and not to either propose a potential system or prepare and propose a detailed roadmap for its development. Further, the report itself was assembled in just a few months at the beginning of 2008 from input by the participants. As such, all inconsistencies reflect either areas where there really are significant open research questions, or misunderstandings by the editor. There was, however, virtually complete agreement about the key challenges that surfaced from the study, and the potential value that solving them may have towards advancing the field of high performance computing.

I am honored to have been part of this study, and wish to thank the study members for their passion for the subject, and for contributing far more of their precious time than they expected.

Peter M. Kogge, Editor and Study Lead
University of Notre Dame
May 1, 2008.

This page intentionally left blank.

Contents

1	Executive Overview	1
2	Defining an Exascale System	5
2.1	Attributes	5
2.1.1	Functional Metrics	5
2.1.2	Physical Attributes	6
2.1.3	Balanced Designs	6
2.1.4	Application Performance	7
2.2	Classes of Exascale Systems	8
2.2.1	Data Center System	8
2.2.2	Exascale and HPC	9
2.2.3	Departmental Systems	9
2.2.4	Embedded Systems	10
2.2.5	Cross-class Applications	11
2.3	Systems Classes and Matching Attributes	12
2.3.1	Capacity Data Center-sized Exa Systems	12
2.3.2	Capability Data Center-sized Exa Systems	13
2.3.3	Departmental Peta Systems	14
2.3.4	Embedded Tera Systems	14
2.4	Prioritizing the Attributes	14
3	Background	17
3.1	Prehistory	17
3.2	Trends	18
3.3	Overall Observations	19
3.4	This Study	19
3.5	Target Timeframes and Tipping Points	20
3.6	Companion Studies	20
3.7	Prior Relevant Studies	21
3.7.1	1999 PITAC Report to the President	21
3.7.2	2000 DSB Report on DoD Supercomputing Needs	21
3.7.3	2001 Survey of National Security HPC Architectural Requirements	21
3.7.4	2001 DoD R&D Agenda For High Productivity Computing Systems	22
3.7.5	2002 HPC for the National Security Community	22
3.7.6	2003 Jason Study on Requirements for ASCI	23
3.7.7	2003 Roadmap for the Revitalization of High-End Computing	23
3.7.8	2004 Getting Up to Speed: The Future of Supercomputing	24

3.7.9	2005 Revitalizing Computer Architecture Research	24
3.7.10	2006 DSB Task Force on Defense Critical Technologies	25
3.7.11	2006 The Landscape of Parallel Computing Research	25
4	Computing as We Know It	27
4.1	Today's Architectures and Execution Models	27
4.1.1	Today's Microarchitectural Trends	27
4.1.1.1	Conventional Microprocessors	28
4.1.1.2	Graphics Processors	28
4.1.1.3	Multi-core Microprocessors	28
4.1.2	Today's Memory Systems	29
4.1.3	Unconventional Architectures	30
4.1.4	Data Center/Supercomputing Systems	31
4.1.4.1	Data Center Architectures	31
4.1.4.2	Data Center Power	32
4.1.4.2.1	Mitigation	33
4.1.4.3	Other Data Center Challenges	33
4.1.5	Departmental Systems	34
4.1.6	Embedded Systems	34
4.1.7	Summary of the State of the Art	35
4.2	Today's Operating Environments	35
4.2.1	Unix	36
4.2.2	Windows NT Kernel	37
4.2.3	Microkernels	37
4.2.4	Middleware	38
4.2.5	Summary of the State of the Art	38
4.3	Today's Programming Models	38
4.3.1	Automatic Parallelization	40
4.3.2	Data Parallel Languages	40
4.3.3	Shared Memory	41
4.3.3.1	OpenMP	42
4.3.3.2	Threads	43
4.3.4	Message Passing	44
4.3.5	PGAS Languages	45
4.3.6	The HPCS Languages	46
4.4	Today's Microprocessors	47
4.4.1	Basic Technology Parameters	47
4.4.2	Overall Chip Parameters	49
4.4.3	Summary of the State of the Art	53
4.5	Today's Top 500 Supercomputers	53
4.5.1	Aggregate Performance	53
4.5.2	Efficiency	54
4.5.3	Performance Components	54
4.5.3.1	Processor Parallelism	55
4.5.3.2	Clock	56
4.5.3.3	Thread Level Concurrency	56
4.5.3.4	Total Concurrency	57
4.5.4	Main Memory Capacity	59

5	Exascale Application Characteristics	61
5.1	Kiviat Diagrams	61
5.2	Balance and the von Neumann Bottleneck	62
5.3	A Typical Application	63
5.4	Exascale Application Characteristics	65
5.5	Memory Intensive Applications of Today	66
5.5.1	Latency-Sensitive Applications	66
5.5.2	Locality Sensitive Applications	68
5.5.3	Communication Costs - Bisection Bandwidth	69
5.6	Exascale Applications Scaling	71
5.6.1	Application Categories	71
5.6.2	Memory Requirements	72
5.6.3	Increasing Non-Main Memory Storage Capacity	73
5.6.3.1	Scratch Storage	73
5.6.3.2	File Storage	73
5.6.3.3	Archival Storage	73
5.6.4	Increasing Memory Bandwidth	74
5.6.5	Increasing Bisection Bandwidth	74
5.6.6	Increasing Processor Count	74
5.7	Application Concurrency Growth and Scalability	75
5.7.1	Projections Based on Current Implementations	75
5.7.2	Projections Based on Theoretical Algorithm Analysis	78
5.7.3	Scaling to Departmental or Embedded Systems	80
5.8	Applications Assessments	81
5.8.1	Summary Observations	81
5.8.2	Implications for Future Research	82
6	Technology Roadmaps	85
6.1	Technological Maturity	86
6.2	Logic Today	87
6.2.1	ITRS Logic Projections	87
6.2.1.1	Power and Energy	88
6.2.1.2	Area	88
6.2.1.3	High Performance Devices	89
6.2.1.4	Low Operating Voltage Devices	89
6.2.1.5	Limitations of Power Density and Its Effect on Operating Frequency	90
6.2.2	Silicon Logic Technology	92
6.2.2.1	Technology Scaling Challenges	92
6.2.2.2	Silicon on Insulator	93
6.2.2.3	Supply Voltage Scaling	94
6.2.2.4	Interaction with Key Circuits	96
6.2.3	Hybrid Logic	97
6.2.4	Superconducting Logic	100
6.2.4.1	Logic Power and Density Comparison	101
6.2.4.1.1	Cooling Costs	102
6.2.4.2	The Memory Challenge	102
6.2.4.3	The Latency Challenge	102
6.2.4.4	The Cross-Cryo Bandwidth Challenge	102

6.3	Main Memory Today	103
6.3.1	The Memory/Storage Hierarchy	103
6.3.2	Memory Types	104
6.3.2.1	SRAM Attributes	104
6.3.2.2	DRAM Attributes and Operation	106
6.3.2.3	NAND Attributes and Operation	107
6.3.2.4	Alternative Memory Types	108
6.3.2.4.1	Phase Change Memory	108
6.3.2.4.2	SONOS Memory	108
6.3.2.4.3	MRAM	109
6.3.3	Main Memory Reliability - Good News	109
6.3.3.1	Trends in FIT Rates	110
6.3.3.2	Immunity to SER	111
6.3.3.3	Possible Issue: Variable Retention Time	111
6.3.3.3.1	Causes	111
6.3.3.3.2	Effects	112
6.3.3.3.3	Mitigation	112
6.3.4	The Main Memory Scaling Challenges	113
6.3.4.1	The Performance Challenge	113
6.3.4.1.1	Bandwidth and Latency	113
6.3.4.1.2	Tradeoffs	113
6.3.4.1.3	Per-pin Limitations	114
6.3.4.2	The Packaging Challenge	114
6.3.4.3	The Power Challenge	115
6.3.4.3.1	Module Power Efficiency	115
6.3.4.3.2	Cell Power	116
6.3.4.4	Major Elements of DRAM Power Consumption	116
6.3.4.4.1	DRAM Operating Modes	117
6.3.4.4.2	DRAM Architecture	118
6.3.4.4.3	Power Consumption Calculations	119
6.3.5	Emerging Memory Technology	120
6.4	Storage Memory Today	122
6.4.1	Disk Technology	122
6.4.1.1	Capacity	123
6.4.1.2	Power	123
6.4.1.3	Transfer Rate and Seek Time	125
6.4.1.4	Time to Move a Petabyte	125
6.4.1.5	Cost	126
6.4.2	Holographic Memory Technology	126
6.4.3	Archival Storage Technology	127
6.5	Interconnect Technologies	127
6.5.1	Strawman Interconnect	128
6.5.1.1	Local Core-level On-chip Interconnect	128
6.5.1.2	Switched Long-range On-chip Interconnect	128
6.5.1.3	Supporting DRAM and CPU Bandwidth	129
6.5.1.4	Intramodule Bandwidth	129
6.5.1.5	Intermodule Bandwidth	129
6.5.1.6	Rack to Rack Bandwidth	129

6.5.2	Signaling on Wire	130
6.5.2.1	Point-to-Point Links	130
6.5.2.1.1	On-Chip Wired Interconnect	130
6.5.2.1.2	Off-chip Wired Interconnect	130
6.5.2.1.3	Direct Chip-Chip Interconnect	131
6.5.2.2	Switches and Routers	132
6.5.3	Optical Interconnects	133
6.5.3.1	Optical Point to Point Communications	134
6.5.3.2	Optical Routed Communications	136
6.5.4	Other Interconnect	137
6.5.5	Implications	137
6.6	Packaging and Cooling	139
6.6.1	Packaging	139
6.6.1.1	Level 1 Packaging	140
6.6.1.2	Level 2 Packaging	141
6.6.2	Cooling	141
6.6.2.1	Module Level Cooling	142
6.6.2.2	Cooling at Higher Levels	144
6.7	System Resiliency	144
6.7.1	Resiliency in Large Scale Systems	145
6.7.2	Device Resiliency Scaling	147
6.7.3	Resiliency Techniques	147
6.7.4	Checkpoint/Rollback	149
6.8	Evolution of Operating Environments	150
6.9	Programming Models and Languages	151
6.9.1	The Evolution of Languages and Models	151
6.9.2	Road map	152
7	Strawmen: Where Evolution Is and Is Not Enough	153
7.1	Subsystem Projections	153
7.1.1	Measurement Units	153
7.1.2	FPU Power Alone	154
7.1.3	Core Energy	155
7.1.4	Main Memory from DRAM	156
7.1.4.1	Number of Chips	156
7.1.4.2	Off-chip Bandwidth	157
7.1.4.3	On-chip Concurrency	157
7.1.5	Packaging and Cooling	158
7.1.6	Non-Main Memory Storage	162
7.1.7	Summary Observations	163
7.2	Evolutionary Data Center Class Strawmen	164
7.2.1	Heavy Node Strawmen	164
7.2.1.1	A Baseline	164
7.2.1.2	Scaling Assumptions	165
7.2.1.3	Power Models	166
7.2.1.4	Projections	166
7.2.2	Light Node Strawmen	170
7.2.2.1	A Baseline	170

7.2.2.2	Scaling Assumptions	172
7.2.2.3	Power Models	173
7.2.2.4	Projections	174
7.3	Aggressive Silicon System Strawman	175
7.3.1	FPU's	176
7.3.2	Single Processor Core	178
7.3.3	On-Chip Accesses	179
7.3.4	Processing Node	181
7.3.5	Rack and System	182
7.3.5.1	System Interconnect Topology	182
7.3.5.2	Router Chips	184
7.3.5.3	Packaging within a rack	184
7.3.6	Secondary Storage	184
7.3.7	An Adaptively Balanced Node	185
7.3.8	Overall Analysis	186
7.3.9	Other Considerations	186
7.3.10	Summary and Translation to Other Exascale System Classes	187
7.3.10.1	Summary: Embedded	189
7.3.10.2	Summary: Departmental	189
7.3.10.3	Summary: Data Center	190
7.4	Exascale Resiliency	190
7.5	Optical Interconnection Networks for Exascale Systems	191
7.5.1	On-Chip Optical Interconnect	192
7.5.2	Off-chip Optical Interconnect	192
7.5.3	Rack to Rack Optical Interconnect	195
7.5.4	Alternative Optically-connected Memory and Storage System	196
7.6	Aggressive Operating Environments	198
7.6.1	Summary of Requirements	198
7.6.2	Phase Change in Operating Environments	199
7.6.3	An Aggressive Strategy	199
7.6.4	Open Questions	200
7.7	Programming Model	202
7.8	Exascale Applications	202
7.8.1	WRF	203
7.8.2	AVUS	203
7.8.3	HPL	204
7.9	Strawman Assessments	204
8	Exascale Challenges and Key Research Areas	207
8.1	Major Challenges	209
8.1.1	The Energy and Power Challenge	209
8.1.1.1	Functional Power	210
8.1.1.2	DRAM Main Memory Power	211
8.1.1.3	Interconnect Power	212
8.1.1.4	Secondary Storage Power	212
8.1.2	The Memory and Storage Challenge	212
8.1.2.1	Main Memory	212
8.1.2.2	Secondary Storage	213

8.1.3	The Concurrency and Locality Challenge	214
8.1.3.1	Extraordinary Concurrency as the Only Game in Town	214
8.1.3.2	Applications Aren't Going in the Same Direction	216
8.1.4	The Resiliency Challenge	217
8.2	Research Thrust Areas	218
8.2.1	Thrust Area: Exascale Hardware Technologies and Architecture	219
8.2.1.1	Energy-efficient Circuits and Architecture In Silicon	220
8.2.1.2	Alternative Low-energy Devices and Circuits for Logic and Memory	222
8.2.1.3	Alternative Low-energy Systems for Memory and Storage	222
8.2.1.4	3D Interconnect, Packaging, and Cooling	223
8.2.1.5	Photonic Interconnect Research Opportunities and Goals	224
8.2.2	Thrust Area: Exascale Architectures and Programming Models	225
8.2.2.1	Systems Architectures and Programming Models to Reduce Communication	225
8.2.2.2	Locality-aware Architectures	225
8.2.3	Thrust Area: Exascale Algorithm and Application Development	227
8.2.3.1	Power and Resiliency Models in Application Models	228
8.2.3.2	Understanding and Adapting Old Algorithms	228
8.2.3.3	Inventing New Algorithms	229
8.2.3.4	Inventing New Applications	229
8.2.3.5	Making Applications Resiliency-Aware	230
8.2.4	Thrust Area: Resilient Exascale Systems	230
8.2.4.1	Energy-efficient Error Detection and Correction Architectures	230
8.2.4.2	Fail-in-place and Self-Healing Systems	230
8.2.4.3	Checkpoint Rollback and Recovery	231
8.2.4.4	Algorithmic-level Fault Checking and Fault Resiliency	231
8.2.4.5	Vertically-Integrated Resilient Systems	231
8.3	Multi-phase Technology Development	232
8.3.1	Phase 1: Systems Architecture Explorations	232
8.3.2	Phase 2: Technology Demonstrators	232
8.3.3	Phase 3: Scalability Slice Prototype	232
A	Exascale Study Group Members	235
A.1	Committee Members	235
A.2	Biographies	235
B	Exascale Computing Study Meetings, Speakers, and Guests	245
B.1	Meeting #1: Study Kickoff	245
B.2	Meeting #2: Roadmaps and Nanotechnology	246
B.3	Special Topics Meeting #1: Packaging	246
B.4	Meeting #3: Logic	247
B.5	Meeting #4: Memory Roadmap and Issues	248
B.6	Special Topics Meeting #2: Architectures and Programming Environments	249
B.7	Special Topics Meeting #3: Applications, Storage, and I/O	250
B.8	Special Topics Meeting #4: Optical Interconnects	250
B.9	Meeting #5: Report Conclusions and Finalization Plans	251
C	Glossary and Abbreviations	253

This page intentionally left blank.

List of Figures

4.1	Three classes of multi-core die microarchitectures.	29
4.2	Microprocessor feature size.	48
4.3	Microprocessor transistor density.	48
4.4	Microprocessor cache capacity.	49
4.5	Microprocessor die size.	50
4.6	Microprocessor transistor count.	50
4.7	Microprocessor V_{dd}	51
4.8	Microprocessor clock.	51
4.9	Microprocessor chip power.	52
4.10	Microprocessor chip power density.	52
4.11	Performance metrics for the Top 10 supercomputers over time.	53
4.12	Efficiency for the Top 10 supercomputers while running Linpack.	54
4.13	Processor parallelism in the Top 10 supercomputers.	55
4.14	Clock rate in the Top 10 supercomputers.	56
4.15	Thread level concurrency in the Top 10 supercomputers.	57
4.16	Total hardware concurrency in the Top 10 supercomputers.	58
4.17	Memory capacity in the Top 10 supercomputers.	58
5.1	Predicted speedup of WRF “Large”.	63
5.2	Time breakdown of WRF by operation category.	64
5.3	Application functionalities.	65
5.4	Predicted speedup of AVUS to latency halving.	66
5.5	Predicted speedup of WRF to latency halving.	67
5.6	Predicted speedup of AMR to latency halving.	67
5.7	Predicted speedup of Hycom to latency halving.	67
5.8	Spatial and temporal locality of strategic applications.	68
5.9	Performance strategic applications as a function of locality.	70
5.10	Growth of communications overhead.	70
5.11	WRF performance response.	75
5.12	AVUS performance response.	76
5.13	HPL performance response.	76
5.14	WRF with $\log(n)$ communications growth.	78
5.15	AVUS with $\log(n)$ communications growth.	79
5.16	Future scaling trends	82
6.1	ITRS roadmap logic device projections	87
6.2	Relative change in key power parameters	90
6.3	Power-constrained clock rate	91

6.4	Technology outlook	92
6.5	Simple transport model	93
6.6	Transistor sub-threshold leakage current and leakage power in recent microprocessors	93
6.7	Technology outlook and estimates	94
6.8	Frequency and power scaling with supply voltage	95
6.9	Sensitivities to changing V_{dd}	96
6.10	Technology scaling, V_t variations, and energy efficiency.	97
6.11	Hybrid logic circuits	99
6.12	CPU and memory cycle time trends.	103
6.13	ITRS roadmap memory density projections.	105
6.14	DRAM cross section.	106
6.15	Programmed and un-programmed NAND cells.	107
6.16	DRAM retention time distribution.	109
6.17	Memory module RMA results.	110
6.18	Variable retention time as it affects refresh distribution.	112
6.19	Industry memory projections.	113
6.20	Reduced latency DRAM.	114
6.21	Center-bonded DRAM package.	114
6.22	Commodity DRAM module power efficiency as a function of bandwidth.	116
6.23	Commodity DRAM voltage scaling.	117
6.24	Block diagram of 1Gbit, X8 DDR2 device.	118
6.25	DDR3 current breakdown for Idle, Active, Read and Write.	119
6.26	Nanoscale memory addressing.	121
6.27	Nanoscale memory via imprint lithography	122
6.28	Disk capacity properties.	123
6.29	Disk power per Exabyte.	124
6.30	Disk transfer rate properties.	124
6.31	Disk price per GB.	125
6.32	Interconnect bandwidth requirements for an Exascale system.	128
6.33	Comparison of 3D chip stacking communications schemes.	131
6.34	Entire optical communication path.	133
6.35	Modulator approach to integrated optics.	135
6.36	Representative current and future high-end level 1 packaging.	139
6.37	Estimated chip counts in recent HPC systems.	144
6.38	Scaling trends for environmental factors that affect resiliency.	146
6.39	Increase in vulnerability as a function of per-socket failure rates.	148
6.40	Projected application utilization when accounting for checkpoint overheads.	149
7.1	Projections to reach an Exaflop per second.	154
7.2	Energy per cycle for several cores.	155
7.3	DRAM as main memory for data center class systems.	156
7.4	Memory access rates in DRAM main memory.	158
7.5	Potential directions for 3D packaging (A).	160
7.6	Potential directions for 3D packaging (B).	161
7.7	Potential directions for 3D packaging (C).	161
7.8	A typical heavy node reference board.	164
7.9	Characteristics of a typical board today.	165
7.10	Heavy node strawman projections.	167

7.11	Heavy node performance projections.	168
7.12	Heavy node GFlops per Watt.	169
7.13	Power distribution in the light node strawman.	172
7.14	Light node strawman performance projections.	174
7.15	Light node strawman Gflops per watt.	175
7.16	Aggressive strawman architecture.	177
7.17	Possible aggressive strawman packaging of a single node.	181
7.18	Power distribution within a node.	182
7.19	The top level of a dragonfly system interconnect.	183
7.20	Power distribution in aggressive strawman system.	186
7.21	Chip super-core organization and photonic interconnect.	192
7.22	Gateway functional block design.	193
7.23	Super-core to super-core optical on-chip link.	194
7.24	Optical system interconnect.	195
7.25	A possible optically connected memory stack.	197
8.1	Exascale goals - Linpack.	208
8.2	Critically of each challenge to each Exascale system class.	209
8.3	The power challenge for an Exaflops Linpack.	211
8.4	The overall concurrency challenge.	215
8.5	The processor parallelism challenge.	216
8.6	Future scaling trends present DARPA-hard challenges.	217
8.7	Sensitivities to changing V_{dd}	221

This page intentionally left blank.

List of Tables

2.1	Attributes of Exascale class systems.	12
2.2	Attributes of Exascale class systems.	15
4.1	Power distribution losses in a typical data center.	32
5.1	Summary applications characteristics.	81
6.1	Some performance comparisons with silicon.	98
6.2	2005 projection of potential RSFQ logic roadmap.	101
6.3	Area comparisons of various memory technologies.	104
6.4	Memory types and characteristics.	108
6.5	Commodity DRAM operating current.	117
6.6	Projected disk characteristics.	122
6.7	Energy budget for optical modulator.	136
6.8	Summary interconnect technology roadmap.	138
6.9	Internal heat removal approaches.	142
6.10	External cooling mechanisms.	143
6.11	Root causes of failures in Terascale systems.	145
6.12	BlueGene FIT budget.	146
7.1	Non-memory storage projections for Exascale systems.	162
7.2	Light node baseline based on Blue Gene.	171
7.3	Summary characteristics of aggressively designed strawman architecture.	176
7.4	Expected area, power, and performance of FPUs with technology scaling.	176
7.5	Expected area, power, and performance of FPUs with more aggressive voltage scaling.	178
7.6	Energy breakdown for a four FPU processor core.	178
7.7	Power budget for strawman multi-core chip.	180
7.8	Area breakdown of processor chip.	181
7.9	Power allocation for adaptive node.	185
7.10	Exascale class system characteristics derived from aggressive design.	188
7.11	Failure rates for the strawman Exascale system.	190
7.12	Checkpointing overheads.	191
7.13	Optical interconnect power parameters.	193
7.14	Optical on-chip interconnect power consumption.	194
7.15	Optical system interconnect power consumption.	196
8.1	The relationship between research thrusts and challenges.	210
A.1	Study Committee Members.	235

Chapter 1

Executive Overview

This report presents the findings and recommendations of the **Exascale Working Group** as conducted over the summer and fall of 2007. The objectives given the study were to understand the course of mainstream computing technology, and determine whether or not it would allow a 1,000X increase in the computational capabilities of computing systems by the 2015 time frame. If current technology trends were deemed as not capable of permitting such increases, then the study was also charged with identifying where were the major challenges, and in what areas may additional targeted research lay the groundwork for overcoming them.

The use of the word “Exascale”¹ in the study title was deliberately aimed at focusing the group’s attention on more than just high end, floating point intensive, supercomputers (“exaflops” machines), but on increasing our ability to perform computations of both traditional and emerging significance at across-the-board levels of performance. The study thus developed as a more precise goal the understanding of technologies to accelerate computing by 1,000X for three distinct classes of systems:

- **data center-sized systems**, where the focus is on achieving 1,000 times the performance of the “petaflop” class systems that will come on line in the next few years, and for more than just numeric intensive applications.
- **departmental-sized systems** that would allow the capabilities of the near-term Petascale machines to be shrunk in size and power to fit within a few racks, allowing widespread deployment.
- **embedded systems** that would allow something approximating a “Terascale” rack of computing such as may be found in the near-term Petascale systems to be reduced to a few chips and a few ten’s of watts that would allow deployment in a host of embedded environments.

Clearly, if done right, a technology path that permits Terascale embedded systems would allow variants of that to serve in Petascale departmental systems, and then in turn grow into larger Exascale supercomputers. The study group recognizes that even though the underlying technologies may be similar, when implementing real systems the actual mix of each technology, and even the architecture of the systems themselves may be different. Thus it was not our focus to design such systems, but to develop a deep understanding of the technological challenges that might prohibit their implementation.

¹The term “Exa” is entering the national dialog through discussions of total Internet bandwidth and on-net storage - see for example *Unleashing the ‘Exaflood’* in the Feb. 22, 2008 Wall Street Journal.

In total, the study concluded that there are four **major challenges** to achieving Exascale systems where current technology trends are simply insufficient, and significant new research is absolutely needed to bring alternatives on line.

- The **Energy and Power Challenge** is the most pervasive of the four, and has its roots in the inability of the group to project any combination of currently mature technologies that will deliver sufficiently powerful systems in any class at the desired power levels. Indeed, a key observation of the study is that it may be easier to solve the power problem associated with base computation than it will be to reduce the problem of transporting data from one site to another - on the same chip, between closely coupled chips in a common package, or between different racks on opposite sides of a large machine room, or on storing data in the aggregate memory hierarchy.
- The **Memory and Storage Challenge** concerns the lack of currently available technology to retain data at high enough capacities, and access it at high enough rates, to support the desired application suites at the desired computational rate, and still fit within an acceptable power envelope. This information storage challenge lies in both main memory (DRAM today) and in secondary storage (rotating disks today).
- The **Concurrency and Locality Challenge** likewise grows out of the flattening of silicon clock rates and the end of increasing single thread performance, which has left explicit, largely programmer visible, parallelism as the only mechanism in silicon to increase overall system performance. While this affects all three classes of systems, projections for the data center class systems in particular indicate that applications may have to support upwards of a billion separate threads to efficiently use the hardware.
- A **Resiliency Challenge** that deals with the ability of a system to continue operation in the presence of either faults or performance fluctuations. This concern grew out of not only the explosive growth in component count for the larger classes of systems, but also out of the need to use advanced technology, at lower voltage levels, where individual devices and circuits become more and more sensitive to local operating environments, and new classes of aging effects become significant.

While the latter three challenges grew out of the high end systems, they are certainly not limited to that class. One need only look at the explosive grow of highly multi-core microprocessors and their voracious appetite for more RAM.

The study's recommendations are thus that significant research needs to be engaged in four major research thrusts whose effects cross all these challenges:

1. Co-development and optimization of **Exascale Hardware Technologies and Architectures**, where the potential of new devices must be evaluated in the context of new architectures that can utilize the new features of such devices to solve the various challenges. This research includes development of energy efficient circuits for communication, memory, and logic; investigation of alternative low-energy devices; development of efficient packaging, interconnection, and cooling technologies; and energy efficient machine organizations. Each of these research areas should be investigated in the context of all Exascale system classes, and metrics should be in terms of energy efficiency realized in this context.
2. Co-development and optimization of **Exascale Architectures and Programming Models**, where the new architectures that arise out of either the device efforts of the previous

thrust or those needed to control the explosive growth in concurrency must be played out against programming models that allow applications to use them efficiently. This work includes developing locality-aware and communication-efficient architectures to minimize energy consumed through data movement, and developing architectures and programming models capable of handling billion-thread concurrency.

3. Co-development of **Exascale Algorithms, Applications, Tools, and Run-times**, where substantial alternatives are needed for programmers to develop and describe, with less than heroic efforts, applications that can in fact use the new architectures and new technology capabilities in efficient enough ways that permit true “Exascale” levels of performance to be sustained.
4. Development of a deep understanding of how to architect **Resilient Exascale Systems**, where the problems in both sheer hardware and algorithmic complexity, and the emergence of new fault mechanisms, are studied in a context that drives the development of circuit, subsystem, and system-level architectures, and the programming models that can exploit them, in ways that allows Exascale systems to provide the kind of dependable service that will make them technically and economically viable.

Further, such research must be very heavily inter-disciplinary. For example, power is the number one concern, but research that focuses only on low-power devices is unlikely to solve the systemic power problems the study encountered. Co-optimization of devices and circuits that employ those devices must be done within the framework of innovative micro and system architectures which can leverage the special features of such devices in the most efficient ways. An explicit and constant attention must be made to **interconnect**, namely the technologies by which one set of functions exchange data with another.

As another example, research that leads to increasing significantly the density of memory parts will clearly help reliability by reducing part count; it may also reduce power significantly by reducing the number of off-chip interfaces that need to be driven, and reduce the number of memory controller circuits that must be incorporated elsewhere.

Finally, if the appropriate technologies are to be developed in time to be considered for 2015 deployments, then, given the level of technical maturity seen by the study group in potential emerging technologies, a three phase research agenda is needed:

1. A **System Architecture Exploration Phase** to not only enhance the maturity of some set of underlying device technologies, but to do enough of the higher level system architecture and modeling efforts to identify how best to employ the technology and what the expected value might be *at the system level*.
2. A **Technology Demonstration Phase** where solutions to the “long poles” in the challenges are developed and demonstrated in an isolated manner.
3. A **Scalability Slice Prototyping Phase** where one or more of the disparate technologies demonstrated in the preceding phase are combined into coherent end-to-end “slices” of a complete Exascale system that permits believable scaling to deployable systems. Such slices are not expected to be a complete system in themselves, but should include enough of multiple subsystems from a potential real system that the transition to a real, complete, system integration is feasible and believable.

The rest of this report is organized as follows:

- Chapter 2 defines the major classes of Exascale systems and their attributes.
- Chapter 3 gives some background to this study in terms of prior trends and studies.
- Chapter 4 discuss the structure and inter-relationship of computing systems today, along with some key historical data.
- Chapter 5 concentrates on developing the characteristics of applications that are liable to be relevant to Exascale systems.
- Chapter 6 reviews the suite of relevant technologies as we understand them today, and develops some roadmaps as to how they are likely to improve in the future.
- Chapter 7 takes the best of known currently available technologies, and projects ahead what such technologies would yield at the Exascale. This is particularly critical in identifying the holes in current technology trends, and where the challenges lie.
- Chapter 8 summarizes the major challenges, develops a list of major research thrusts to answer those challenges, and suggests a staged development effort to bring them to fruition.

Chapter 2

Defining an Exascale System

The goal of this study was not to design Exascale systems, but to identify the overall challenges and problems that need special emphasis between now and 2010 in order to have a technology base sufficient to support development and deployment of Exascale-class systems by 2015. However, to do this, it is first necessary to define more carefully those classes of systems that would fall under the Exascale rubric. In this chapter, we first discuss the attributes by which achievement of the label “Exascale” may be claimed, then the classes of systems that would lever such attributes into “Exascale” systems.

2.1 Attributes

To get to the point of being able to analyze classes of Exascale systems, we first need a definition of what “Exascale” means. For this study, an Exascale system is taken to mean that one or more key attributes of the system has 1,000 times the value of what an attribute of a “Petascale” system of 2010 will have. The study defined three dimensions for these attributes: functional performance, physical attributes, and application performance. Each is discussed below.

2.1.1 Functional Metrics

We define a functional metric as an attribute of a system that directly measures some parameter that relates to the ability of the system to solve problems. The three major metrics for this category include:

- **Basic computational rate:** the rate at which some type of operation can be executed per second. This includes, but is not limited to:
 - **flops:** floating point operations per second.
 - **IPS:** instructions per second.
 - and (remote) **memory accesses** per second.
- **Storage capacity:** how much memory is available to support holding different forms of the problem state at different times. Specific metrics here include the capacities of various parts of the storage hierarchy:
 - **Main memory:** the memory out of which an application program can directly access data via simple loads and stores.

- **Scratch Storage:** the memory needed to hold checkpoints, I/O buffers, and the like during a computation.
 - **Persistent Storage:** the storage to hold both initial data sets, parameter files, computed results, and long-term data used by multiple independent runs of the application.
- **Bandwidth:** the rate at which data relevant to computation can be moved around the system, usually between memory and processing logic. Again, a variety of specialized forms of bandwidth metrics are relevant, including:
 - **Bisection bandwidth:** for large systems, what is the bandwidth if we partition the system in half, and measure the maximum that might flow from one half of a system to the other.
 - **Local memory bandwidth:** how fast can data be transferred between memories and closest computational nodes.
 - **Checkpoint bandwidth:** how fast copies of memory can be backed up to a secondary storage medium to permit roll-back to some fixed point in the program if a later error is discovered in the computations due to hardware fault.
 - **I/O bandwidth:** the rate at which data can be extracted from the computational regions of the system to secondary storage or visualization facilities where it can be saved or analyzed later.
 - **On chip bandwidth:** how fast can data be exchanged between functional units and memory structures within a single chip.

2.1.2 Physical Attributes

The second class of attributes that are relevant to an Exascale discussion are those related to the instantiation of the design as a real system, primarily:

- Total power consumption
- Physical size (both area and volume)
- Cost

Since this is primarily a technology study, cost is one metric that will for the most part be ignored.

While peak values for all the above are important, from a technology vantage, it proved to be more relevant to use these metrics as denominators for ratios involving the functional metrics described above. Further, for power it will also be valuable to focus not just on the “per watt” values of a technology, but a “per joule” metric. Knowing the rates per joule allows computation of the total energy needed to solve some problem; dividing by the time desired gives the total power.

2.1.3 Balanced Designs

While individual metrics give valuable insight into a design, more complete evaluations occur when we consider how designs are **balanced**, that is how equal in efficiency of use is the design with respect to different high-cost resources. This balance is usually a function of application class.

For example, some systems based on special purpose processor designs such as Grape[97] may be “well-balanced” for some narrow classes of applications that consume the whole machine (such as multi-body problems), but become very inefficient when applied to others (insufficient memory, bandwidth, or ability to manage long latencies). Other systems such as large scale search engines may be built out of more commodity components, but have specialized software stacks and are balanced so that they perform well only for loads consisting of large numbers of short, independent, queries that interact only through access to large, persistent databases.

Classically, such balance discussions have been reduced to simple ratios of metrics from above, such as “bytes to flops per second” or “Gb/s per flops per second.” While useful historical vignettes, care must be taken with such ratios when looking at different scales of applications, and different application classes.

2.1.4 Application Performance

There are two major reasons why one invests in a new computing system: for solving problems not previously solvable, either because of time to solution or size of problem, or for solving the same kinds of problems solved on a prior system, but faster or more frequently. Systems that are built for such purposes are known as capability and capacity systems respectively. The NRC report *Getting Up to Speed* ([54] page 24) defines these terms more formally:

The largest supercomputers are used for **capability** or turnaround computing where the maximum processing power is applied to a single problem. The goal is to solve a larger problem, or to solve a single problem in a shorter period of time. Capability computing also enables the solution of problems that cannot otherwise be solved in a reasonable period of time (for example, by moving from a two-dimensional to a three-dimensional simulation, using finer grids, or using more realistic models). Capability computing also enables the solution of problems with real-time constraints (e.g. intelligence processing and analysis). The main figure of merit is time to solution.

Smaller or cheaper systems are used for **capacity** computing, where smaller problems are solved. Capacity computing can be used to enable parametric studies or to explore design alternatives; it is often needed to prepare for more expensive runs on capability systems. Capacity systems will often run several jobs simultaneously. The main figure of merit is sustained performance per unit cost.

Capacity machines are designed for throughput acceleration; they accelerate the rate at which certain types of currently solvable applications can be solved. Capability machines change the spectrum of applications that are now “solvable,” either because such problems can now be fit in the machine and solved at all, or because they can now be solved fast enough for the results to be meaningful (such as weather forecasting).

We note that there is significant fuzziness in these definitions, especially in the “real-time” arena. A machine may be called a capability machine if it can take a problem that is solvable in prior generations of machines (but not in time for the results to be useful), and make it solvable (at the same levels of accuracy) in some fixed period of time where the results have value (such as a weather forecasting). A machine that solves the same problems even faster, or solves multiple versions concurrently, and still in real-time, may be called a capacity machine. It is not uncommon for today’s capability systems to become tomorrow’s capacity systems and newer, and even more capable, machines are introduced.

Also, some machines may have significant aspects of both, such as large scale search engines, where the capability part lies in the ability to retain significant indexes to information, and the capacity part lies in the ability to handle very large numbers of simultaneous queries.

Although there is significant validity to the premise that the difference between a capacity and a capability machine may be just in the “style” of computing or in a job scheduling policy, we will continue the distinction here a bit further mainly because it may affect several system parameters that in turn affect real hardware and related technology needs. In particular, it may relate to the breadth of applications for which a particular design point is “balanced” to the point of being economically valuable. As an example, it appears from the strawmen designs of Chapter 7 that silicon-based Exascale machines may be significantly memory-poor in relation to today’s supercomputing systems. This may be acceptable for some large “capability” problems where the volume of computation scales faster than required data size, but not for “capacity” applications where there is insufficient per-node memory space to allow enough copies of enough different data sets to reside simultaneously in the machine in a way that can use all the computational capabilities.

2.2 Classes of Exascale Systems

To reiterate the purpose of this study, it is to identify those technology challenges that stand in the way of achieving initial deployment of Exascale systems by 2015. The approach taken here for identifying when such future systems may be called “Exascale” is multi-step. First, we partition the Exascale space based on the gross physical characteristics of a deployed system. Then, we identify the functional and application performance characteristics we expect to see needed to achieve “1,000X” increase in “capability.” Finally, we look at the ratios of these characteristics to power and volume as discussed above. Then by looking at these ratios we can identify where the largest challenges are liable to arise, and which class of systems will exhibit them.

In terms of overall physical size and power, we partition the 2015 system space into three categories:

- “Exa-sized” data center systems,
- “Peta-sized” departmental computing systems,
- and “Tera-sized” Embedded systems.

It is very important to understand that these three systems are not just a 1,000X or 1,000,000X scaling in all parameters. Depending on the application class for each level, the individual mix of parameters such as computational rates, memory capacity, and memory bandwidth may vary dramatically.

2.2.1 Data Center System

For this study, an exa-sized **data center system** of 2015 is one that roughly corresponds to a typical notion of a supercomputer center today - a large machine room of several thousand square feet and multiple megawatts of power consumption. This is the class system that would fall in the same footprint as the Petascale systems of 2010, except with 1,000X the capability. Because of the difficulty of achieving such physical constraints, the study was permitted to assume some growth, perhaps a factor of 2X, to something with a maximum limit of 500 racks and 20 MW for the computational part of the 2015 system.

2.2.2 Exascale and HPC

In **high-end computing** (i.e. **supercomputing** or **high-performance computing**), the major milestones are the emergence of systems whose aggregate performance first crosses a threshold of 10^{3k} operations performed per second, for some k . Gigascale (10^9) was achieved in 1985 with the delivery of the Cray 2. Terascale (10^{12}) was achieved in 1997 with the delivery of the Intel ASCI Red system to Sandia National Laboratory. Today, there are contracts for near-Petascale (10^{15}) systems, and the first will likely be deployed in 2008. Assuming that progress continues to accelerate, one might hope to see an Exascale (10^{18}) system as early as 2015.

For most scientific and engineering applications, Exascale implies 10^{18} IEEE 754 Double Precision (64-bit) operations (multiplications and/or additions) per second (**exaflops**¹). The **High Performance Linpack (HPL)** benchmark[118], which solves a dense linear system using LU factorization with partial pivoting, is the current benchmark by which the community measures the throughput of a computing system. To be generally accepted as an Exascale system, a computer must exceed 10^{18} flops (1 exaflops) on the HPL benchmark. However, there are critical Defense and Intelligence problems for which the operations would be over the integers or some other number field. Thus a true Exascale system has to execute a fairly rich instruction set at 10^{18} operations per second lest it be simply a special purpose machine for one small family of problems.

A truly general purpose computer must provide a balance of arithmetic throughput with memory volume, memory and communication bandwidth, persistent storage, etc. To perform the HPL benchmark in a reasonable period of time, an Exascale system would require on the order of 10 petabytes (10^{16} Bytes) of main memory. Such a system could credibly solve a small set of other problems. However, it would need at least another order-of-magnitude of additional main memory (10^{17} bytes) to address as broad a range of problems as will be tackled on near-Petascale systems in the next year or so. Amdahl's rule of thumb was one byte of main memory per operation, but in recent years systems have been deployed with less (0.14 to 0.3), reflecting both the diverging relative costs of the components as well as the evolving needs of applications. Chapter 5 tries to get a better handle on this as we scale algorithms up.

Finally, an Exascale system must provide data bandwidth to and from at least local subsets of its memory at approximately an exabyte per second. Section 5.6 discusses sensitivities of many current algorithms to bandwidth.

To store checkpoints, intermediate files, external data, results, and to stage jobs, an Exascale system will need at least another order-of-magnitude (i.e., 10^{18} Bytes) of persistent storage, analogous to today's disk arrays, with another 10-100X for file storage. Section 5.6.3 explores these numbers in more detail.

2.2.3 Departmental Systems

The discussion in the previous was oriented towards the requirements for a leadership-class Exascale system deployed to support a handful of national-scale, capability jobs. To be economically viable, the technology developed for such a system must also be utilized in higher volume systems such as those deployed for departmental-scale computing in industry and government. To the casual eye, the biggest difference will be in the physical size of the departmental systems, which will only fill a few racks. Power density will be a critical aspect of these systems, as many customers will want them to be air cooled. Others may only have building chilled water available. Thus, a petasized **departmental system** of 2015 would be one whose computational capabilities would match

¹We denote a computation involving a total of 10^{18} floating point operations as an **exaflop** of computation; if they are all performed in one second then the performance is one **exaflops**.

roughly those of a 2010 Petascale data center-sized system, but in the form factor of a departmental computing cluster - perhaps one or two racks, with a maximum power budget of what could be found in reasonable computer “cluster” room of today - perhaps 100-200KW at maximum. In a sense this is around 1/1,000th in both capability and size of the larger systems.

The principle differentiation between the Exascale data center systems and departmental systems will be the need to support a large set of third-party applications. These span a broad range from business applications like Oracle to engineering codes like LS-DYNA[32]. These are very large, sophisticated applications that demand the services of a full featured operating system, usually a derivative of UNIX, not a micro-kernel as may suffice on the early Exascale systems. The operating system will need to support virtualization and to interact with a **Grid**[45] of external systems as the department will likely be part of a much larger, geographically distributed enterprise.

To support a broad range of such mainstream applications, departmental systems composed of Exascale components will require a proportionately larger main memory and persistent store. It’s not uncommon today to see systems such as the SGI Altix delivered with only four Intel Itanium CPUs yet a terabyte of main memory. Latency and bandwidth to the memory hierarchy (DRAM and disk) is already a problem today for mainstream applications, and probably cannot be tapered as aggressively as it likely will be on a leadership, Exascale system.

The third party applications whose availability will be critical to the success of departmental scale systems in 2015 run on up to 100 processors today. Only a handful of scaling studies or heroic runs exceed that. Petascale departmental systems will likely have 10^5 , perhaps even 10^6 threads in them. Extending enough of today’s applications to this level of parallelism will be a daunting task. In the last fifteen years, commercial software developers have struggled to transition their codes from one CPU (perhaps a vector system) to $O(1000)$ today. They will have to expand this scalability by another three orders-of-magnitude in the next decade. This will require breakthroughs not only in computer architecture, compilers and other software tools, but also in diverse areas of applied mathematics, science, and engineering. Not amount of system concurrency can overcome an algorithm that does not scale well. Finally, mainstream applications can not be expected to have evolved to the point where they can adapt to faults in the underlying system. Thus mean-time-to-failure comparable to today’s large-scale servers will be necessary.

2.2.4 Embedded Systems

One of the motivations for the development of parallel computing systems in the late 1980’s and early 1990’s was to exploit the economies of scale enjoyed by the developers of mainstream, commodity components. At the time, commodity meant personal computers and servers. Today, the volume of electronics created for embedded systems, such as cell phones, dwarfs desktop and servers, and this trend will likely continue. Thus its increasingly clear that high-end computing will need to leverage embedded computing technology. In fact, the blending of embedded technology and HPC has already begun, as demonstrated by IBM’s Blue Gene family of supercomputers, and HPC derivatives, such as the Roadrunner at Los Alamos National Labs, using the STI Cell chip (which has its origins in game systems).

If one hopes to use Exascale technology in embedded systems, not just leadership and departmental scale systems, then there are a number of additional issues that must be addressed. The first is the need to minimize size, weight, and power consumption. While embedded systems are often very aggressive in their use of novel packaging technology (e.g. today’s DRAM stacks), they cannot similarly exploit the most aggressive liquid cooling technology which is often available in large machine rooms.

Historically embedded processors traded lower performance for greater power efficiency, whereas

processors designed for large servers maximized performance, and power was only constrained by the need to cool the system. At Exascale, this will no longer be the case. As this study documents, limiting power consumption will be a major issue for Exascale systems. Therefore, this major difference in the design of embedded and server systems may very well disappear. This is not to say that the two processor niches will merge. Servers will need to be engineered as components in much larger systems, and will have to support large address spaces that will not burden embedded systems.

Embedded systems, especially those developed for national security applications, often operate in hostile environment conditions for long periods of time, during which they cannot be serviced. Space is the obvious example, and to be useful in such an environment, one must be able to extend Exascale technology to be radiation hard. The architecture must be fault tolerant, and able to degrade gracefully when individual components inevitably fail. It is interesting to speculate that this distinction between the fault tolerance design standards for today's embedded vs. enterprise systems may disappear as we approach Exascale since smaller geometries and lower voltage thresholds will lead to components that are inherently less reliable even when deployed in normal, well controlled environments.

As with departmental scale systems, the biggest difference between pioneering embedded Exascale systems and their contemporary embedded systems will likely be the software. Whereas most scientific and engineering applications operate on double precision values, embedded applications are often fixed point (there is no need to carry more precision than the external sensors provide) or, increasingly, single precision. The applications and hence the operating system will often have hard real time constraints. The storage required to hold application is often limited, and this bounds the memory footprint of the operating systems and the application.

2.2.5 Cross-class Applications

While the three classes were discussed above in isolation, there are significant applications that may end up needing all three at the same time. Consider, for example, a unified battlefield **persistent surveillance** system that starts with drones or other sensor platforms to monitor a wide spectrum of potential information domains, proceeds through local multi-sensor correlation for real-time event detection, target development, and sensor management to post-event collection management and forensic analysis, where tactical and strategic trends can be identified and extracted.

A bit more formally, a recent DSB report[18] (page 103) defines persistent surveillance as follows:

The systematic and integrated management of collection processing, and customer collaboration for assured monitoring of all classes of threat entities, activities and environments in physical, aural or cyber space with sufficient frequency, accuracy, resolution, precision, spectral diversity, spatial extent, spatial and sensing diversity and other enhanced temporal and other performance attributes in order to obtain the desired adversary information, even in the presence of deception.

Implementing such systems may very well require Exascale embedded systems in a host of different platforms to do radar, video, aural, or cyber processing, analysis, and feature extraction; Exascale departmental systems may be needed to integrate multiple sensors and perform local event detection and sensor management, and larger Exascale data center class systems are needed for overall information collection and deep analysis.

	Attributes				
	Aggregate Computational Rate	Aggregate Memory Capacity	Aggregate Bandwidth	Volume	Power
Exa Scale Data Center Capacity System relative to 2010 Peta Capacity System					
Single Job Speedup	1000X flops	Same	1000X	Same	Same
Job Replication	1000X flops	up to 1000X	1000X	Same	Same
Exa Scale Data Center Capability System relative to 2010 Peta Capability System					
Current in Real-Time	1000X flops, ops	Same	1000X	Same	Same
Scaled Current Apps	up to 1000X flops, ops	up to 1000X	up to 1000X	Same	Same
New Apps	up to 1000X flops, ops, mem accesses	up to 1000X - with more persistence	up to 1000X	Same	Same
Peta Scale Department System relative to 2010 Peta HPC System					
	Same	Same	Same	1/1000	1/1000
Tera Scale Embedded System relative to 2010 Peta HPC System					
	1/1000	1/1000	1/1000	1/1 million	1/1 million

Table 2.1: Attributes of Exascale class systems.

2.3 Systems Classes and Matching Attributes

The following subsections discuss each of these target system, with Table 2.1 summarizing how the various metrics might inter-relate. The header row for each class includes a reference to the type of system used as a baseline in estimating how much on an increases in the various attributes are needed to achieve something that would be termed an Exascale system. Attribute columns with the value “same” should be interpreted as being close to (i.e. perhaps within a factor of 2) of the same numbers for the 2010 reference point.

2.3.1 Capacity Data Center-sized Exa Systems

A 2015 data center sized capacity system is one whose goal would be to allow roughly 1,000X the production of results from the same applications that run in 2010 on the Petascale systems of the time. In particular, by “same” is meant “approximately” (within a factor of 2 or so) the same data set size (and thus memory capacity) and same application code (and thus the same primary metric of computational rate as for the reference 2010 Petascale applications - probably still flops).

There are two potential variants of this class, based on how the increase in throughput is achieved: by increasing the computational rate as applied to a single job by 1,000X, or by concurrently running multiple (up to 1,000) jobs through the system at the same time. In either case the aggregate computational rates must increase by 1,000X. For the single job speedup case, bandwidths probably scale linearly with the 1,000X in rate, but total memory capacity will stay roughly flat at what it will be in 2010.

For systems supporting concurrent jobs, the total memory capacity must scale with the degree of concurrency. While most of the lower levels of bandwidth must scale, it may be that the system-

wide bandwidth need not scale linearly, since the footprint in terms of racks needed for each job may also shrink.

2.3.2 Capability Data Center-sized Exa Systems

A 2015 data center sized capability system is one whose goal would be to allow solution of problems up to 1,000 times “more complex” than solvable by a 2010 peta capability system. In contrast to capacity machines, these systems are assumed to run only one application at a time, so that sizing the various attributes need reflect only one application. There are at least three variants of such systems:

- Real-time performance: where the same application that runs on a peta system needs to run 1,000X faster to achieve value for predictive purposes. As with the capacity system that was speedup based, computational rate and bandwidth will scale, but it may be that memory capacity need not.

In this scenario, if the basic speed of the computational units does not increase significantly (as is likely), then new levels of parallelism must be discovered in the underlying algorithms, and if that parallelism takes a different form than the current coarse-grained parallelism used on current high end systems, then the software models will have to be developed to support that form of parallelism.

- Upscaling of current peta applications: where the overall application is the same as exists for a 2010 peta scale system, but the data set size representing problems of interest needs to grow considerably. There the meaning of a 1000X increase in computing may have several interpretations. A special case is the weak-scaling scenario in which a fixed amount of data is used per thread; if the computation is linear time in the data size, then this corresponds to a 1000x increase in memory capacity along with computation and bandwidth. Another obvious one may mean solving problem sizes that require 1,000X the number of computations in the same time as the maximum sized problems did on the reference peta system. Here computational and bandwidths scale by 1,000X, and memory must scale at least as fast enough to contain the size problems that force the 1,000X increase in computation. This memory scaling may thus vary from almost flat to linear in the performance increase. Intermediate numbers are to be expected, where, for example, 4D simulations may have an $N^{3/4}$ law, meaning that 1,000X in performance requires $1000^{3/4} = 178X$ the memory capacity.

A second interpretation of a 1000X gain in computation is that there may be some product between increase in performance and increase in problem size (i.e. storage). Thus, for example, a valid definition of an Exascale system may be one that supports data sets 100X that of today, and provides 10X more computation per second against its solution, regardless of how long the total problem execution takes.

- New applications: where the properties of the desired computation looks nothing like what is supportable today. This includes the types of operations that in the end dominate the computational rate requirements (instead of flops, perhaps integer ops or memory accesses), the amount and type of memory (very large graphs, for example, that must persist in memory essentially continuously), to bandwidth (which might even explode to become the dominant performance parameter). These new application might very well use algorithms that are unknown today, along with new software and architecture models.

2.3.3 Departmental Peta Systems

Another version of an exa sized system might be one that takes a 2010 peta scale system (encompassing 100s' of racks) and physically reduces it to a rack or two that may fit in, and be dedicated to, the needs of a small part of an organization, such as a department. This rack would thus in 2015 have the computational attributes of a 2010 peta system in the footprint of a 2010 tera scale systems.

If the goal is to shrink a 2010 peta scale system, then the overall computational rate and memory capacity at an absolute level, are unchanged from the 2010 numbers. The power and volume limits, however, must scale by a factor of about 1/1000. Also, while the aggregate internal system bandwidth is unchanged from the 2010 system, reducing the physical size also means that where the bandwidth requirements must be met changes. For example, going to a 1-2 rack system means that much more of the 2015 inter-rack data bandwidth must be present within a single rack.

2.3.4 Embedded Tera Systems

The final exa technology-based system in 2015 would be a system where the computational potential of a 2010 tera scale system is converted downwards to a few chips and a few tens of watts. This might be the basis for a tera scale workstation of PC, or a tera scale chip set for embedded applications. The latter has the most relevance to DoD missions, and is least likely to be achieved by commercial developments, so it will be the "low end" of the Exascale systems for the rest of this report.

Much of the discussion presented above for the departmental sized system is still relevant, but with all the bandwidth now supported between a handful of chips.

If we still reference a peta scale system as a starting point, this means that the key attributes of rate, capacity, and bandwidth all decrease by a factor of 1,000, and the volume and power ratios by a factor of 1 million. This, however, may not be very precise, since the application suite for a tera-sized system that fits in say an aircraft will not be the same suite supported by a Petascale system. This is particularly true for memory, where scaling down in performance may or may not result in a different scaling down in memory capacity

2.4 Prioritizing the Attributes

If as is likely, achieving factors of 1,000X in any particular dimension are liable to be the most difficult to achieve, then looking at which systems from the above suite have the most 1,000X multipliers is liable to indicate both which system types might be the most challenging, and which attributes are liable to be the most relevant to the greatest number of systems.

Table 2.2 summarizes the metrics from the prior chart when they are ratioed for "per watt" and "per unit volume." At the bottom and on the left edge the number of attribute entries that are "1,000X" are recorded. From these, it is clear that the capability, departmental, and embedded systems seem to have the most concurrent challenges, and that the computational rate and bandwidth "per watt" and "per volume" are uniformly important. The only reason why memory capacity doesn't rise to the same level as these other two is that we do not at this time fully understand the scaling rules needed to raise problems up to the exa level, although the analysis of Chapter 5 indicates that significantly more than is likely in our strawmen of Chapter 7 is likely.

Regardless of this analysis, however, the real question is how hard is it to achieve any of these ratios, for any of the systems, in the technologies that may be available.

	Attributes						# of 1000X Ratios
	Comp. Rate	Memory Cap.	BW	Comp. Rate	Memory Cap.	BW	
Exa Scale Data Center Capacity System relative to 2010 Peta Capacity System							
Single Job Speedup	1000X	Same	1000X	1000X	Same	1000X	4
Job Replication	1000X	up to 1000X	1000X	1000X	up to 1000X	1000X	6
Exa Scale Data Center Capability System relative to 2010 Peta Capability System							
Current in Real-Time	1000X	Same	1000X	1000X	Same	1000X	4
Scaled Current Apps	up to 1000X	up to 1000X	up to 1000X	up to 1000X	up to 1000X	up to 1000X	6
New Apps	up to 1000X	up to 1000X	up to 1000X	up to 1000X	up to 1000X	up to 1000X	6
Peta Scale Department System relative to 2010 Peta HPC System							
	1000X	1000X	1000X	1000X	1000X	1000X	6
Tera Scale Embedded System relative to 2010 Peta HPC System							
	1000X	1000X	1000X	1000X	1000X	1000X	6
# of 1000X ratios	7	5	7	7	5	7	

Table 2.2: Attributes of Exascale class systems.

This page intentionally left blank.

Chapter 3

Background

This chapter provides a bit of background about trends in leading edge computational systems that are relevant to this work.

3.1 Prehistory

In the early 1990s, it became obvious that there was both the need for, and the potential to, achieve a trillion (10^{12}) floating point operations per second against problems that could be composed by dense linear algebra. This reached reality in 1996 when the ASCI Red machine passed 1 **teraflops**¹ (a teraflop per second) in both peak² and sustained³ performance.

In early 1994, more than two years before the achievement of a teraflops, an effort was started[129] to define the characteristics of systems (the device technologies, architectures, and support software) that would be needed to achieve one thousand times a teraflops, namely a **petaflops** - a million billion floating point operations per second, and whether or not there were real and significant applications that could take advantage of such systems. This effort triggered multiple government-sponsored research efforts such as the **HTMT (Hybrid Technology Multi-Threaded)**[47] and **HPCS (High Productivity Computing Systems)**[4] programs, which in turn helped lead to peak petaflops systems within a year of now, and sustained petaflops-level systems by 2010.

At the same time as this explosion at the high end of “classical” scientific computing occurred, an equally compelling explosion has occurred for applications with little to do with such floating point intensive computations, but where supporting them at speed requires computing resources rivaling those of any supercomputer. Internet commerce has led to massive **server systems** with thousands of processors, performing highly concurrent web serving, order processing, inventory control, data base processing, and data mining. The explosion of digital cameras and cell phones with rich multi-media capabilities have led to growing on-line, and heavily linked, object data bases with the growing need to perform real-time multimedia storage, searching, and categorization. Intelligence applications after 9/11 have developed to anticipate the actions of terrorists and rogue states.

Thus in a real sense the need for advanced computing has grown significantly beyond the need for just flops, and to recognize that fact, we will use the terms **gigascale**, **Terascale**, **Petascale**, etc to reflect such systems.

¹In this report we will define a **gigaflop**, **teraflop**, etc to represent a billion, trillion, etc floating point operations. Adding an “s” to the end of such terms will denote a “per second” performance metric

²**Peak performance** is a measure of the maximum concurrency in terms of the maximum number of relevant operations that the hardware could sustain in any conditions in a second

³**Sustained performance** is a measure of the number of relevant operations that a real application can execute on the hardware per second

In addition to this drive for high-end computing, equally compelling cases have been building for continually increasing the computational capabilities of “smaller” systems. Most of the non “top 10” of the “Top 500” systems are smaller than world-class systems but provide for an increasing base of users the performance offered by prior years supercomputing. The development of **server blades** has led to very scalable systems built out of commercial microprocessor technology. In a real sense, updates to the prior generation of Terascale computing has led to the Petascale computing of the high end, which in turn is becoming the technology driver for the next generation of broadly deployed Terascale computing.

In the embedded arena, the relentless push for more functionality such as multi-media, video, and GPS processing, in smaller packages (both for personal, industrial and scientific uses) has become a linchpin of our modern economy and defense establishment, with a growing need to allow such packages to function in a growing suite of difficult environmental conditions. Thus just as we have moved to gigascale laptops, PDAs, etc with today’s technologies, the trend is clearly to migrate into Terascale performance in similar sizes.

3.2 Trends

At the same time the above events were happening, a series of trends have emerged that has cast some doubt as to the ability of “technology development as usual” to provide the kind of leap that drove Terascale and Petascale computing. These include:

- Moore’s Law, if (correctly) interpreted as doubling the number of devices per unit of area on a chip every 18-24 months, will continue to do so.
- Moore’s Law, if (incorrectly) interpreted as of doubling performance every 24 months, has hit a power wall, where clock rates have been essentially flat since the early 2000s.
- Our ability to automatically extract from serial programs more operations out of normal programs to perform in parallel in the hardware has plateaued.
- Our ability to hide the growing memory latency wall by increasingly large and complex cache hierarchies has hit limits in terms of its effectiveness on real applications.
- Memory chip designs are now being driven by the need to support large amounts of very cheap nonvolatile memory, ideally with medium high bandwidth but where read and write latencies are almost irrelevant.
- Our ability to devise devices with finer and finer feature sizes is being limited by lithography, which seems to be stuck for the foreseeable future with what can be formed from 193 nm deep ultraviolet light sources.
- The cost of designing new microprocessors in leading edge technologies has skyrocketed to the point where only a very few large industrial firms can afford to design new chips.
- Funding for new research in computer architecture to look for alternatives to these trends has declined to the point where senior leaders in the field have raised alarms[100], and major research communities organized studies outlining the problem and why it should be addressed[12]. The traditional model of single-domain research activities where hardware and software techniques are explored in isolation will not address the current challenges.

3.3 Overall Observations

While it took something approaching 16 years to get from the first serious discussions of Petascale computers (1994) to their emergence as deployable systems (expected in 2010), the technologies, architectures, and programming models were all remarkably foreseen early on[129], and in retrospect bear a real family resemblance to the early Terascale machines. Looking forward to another similar three-order jump in computational capabilities (termed here as **Exascale** systems), several observations emerge:

- There is a continuing need for critical applications to run at much higher rates of performance than will be possible even with the Petascale machines.
- These applications are evolving into more complex entities than those from the Terascale era.
- This need for increasing computing capability is more than just as absolute numbers, but also for reducing the size of the systems that perform today's levels of computing into smaller and smaller packages.
- Technology is hitting walls for which there is no visible viable solutions, and commercial pressures are not driving vendors in directions that will provide the advanced capabilities needed for the next level of performance.
- There is a dearth of advanced computer architecture research that can leverage either existing or emerging technologies in ways that can provide another explosive growth in performance.
- Our programming methodologies have evolved to the point where with some heroics scientific codes can be run on machines with tens' of thousands of processors, but our ability to scale up applications to the millions of processors, or even port conventional "personal" codes to a few dozen cores (as will be available soon in everyone's laptops) is almost non-existent.

Given this, and given the clear need for deploying systems with such capabilities, it seems clear that without explicit direction and support, even another 16 years of waiting cannot guarantee the emergence of Exascale systems.

3.4 This Study

The project that led to this report was started by DARPA in the spring of 2007, with the general objective of assisting the U.S. government in exploring the issues, technical limitations, key concepts, and potential enabling solutions for deploying Exascale computing systems, and ensuring that such technologies are mature enough and in place by 2015, not a decade later.

A contract was let to the Georgia Institute of Technology to manage a study group that would cover the technological spectrum. A study chair, Dr. Peter Kogge of the University of Notre Dame, was appointed, and a group of nationally recognized experts assembled (Appendix A). The study group's charter was to explore the issues, technical limitations, key concepts, and potential solutions to enable Exascale computing by addressing the needed system technologies, architectures, and methodologies.

The key outcome of the study was to develop an open report (this document) that identifies the key roadblocks, challenges, and technology developments that must be tackled by 2010 to support potential 2015 deployment. A subsidiary goal was to generate ideas that will permit use of similar technologies for Terascale embedded platforms in the same time frame.

Over the summer and fall of 2007, almost a dozen meetings were held at various venues, with a large number of additional experts invited in for discussions on specific topics (Appendix B).

3.5 Target Timeframes and Tipping Points

A target time for the development of Exascale technologies and systems was chosen as 2015. This was chosen as an aggressive goal because it represented about 1/2 of the time required to get to Petascale from the first workshop in 1994, and mirrors the approximate time period required to formulate and execute the HPCS program from start to finish.

This target of 2015 included both technologies and systems as goals. Achieving the former would mean that all technologies would be in place by 2015 to develop and deploy any class of Exascale system. Achieving the latter means that systems of some, but not necessarily all, Exascale classes (perhaps embedded or departmental) with Exascale levels of capabilities (as defined here) would be possible.

For such 2015 deployments to be commercially feasible, the underlying technologies must have been available long enough in advance for design and development activities to utilize them. Thus, a secondary time target was placed at 2013 to 2014 for base technologies out of which product components could be designed.

Finally, in looking back at the Petascale development, even though there was a significant effort invested in the early years, it wasn't until the early 2000s' that it became clear to commercial vendors that Petascale systems were in fact going to be commercially feasible, even if it wasn't obvious what the architecture or exact designs would be. We refer to such a time as the **tipping point** in terms of technology development, and thus set as a study goal the task of defining what emerging technologies ought be pushed so that an equivalent tipping point in terms of their potential usefulness for Exascale systems would be reached by 2010. At this point, it is hoped that industry can see clearly the value of such technologies, and can begin to pencil in their use in systems by 2015.

3.6 Companion Studies

Concurrent with this study were two other activities with significant synergism. First is a DoE-sponsored study entitled "Simulation and Modeling at the Exascale for Energy, Ecological Sustainability and Global Security (E3SGS)"⁴, whose goal is to set the stage for supercomputer-class Exascale applications that can attack global challenges through modeling and simulation. A series of town-hall meetings was held in order to develop a more complete understanding of the properties of such applications and the algorithms that underly them.

These meetings focused primarily on Exascale applications and possible algorithms needed for their implementation, and not on underlying system architecture or technology.

Second is a workshop held in Oct. 2007 on "Frontiers of Extreme Computing," which represented the third in a series of workshops⁵ on pushing computation towards some sort of ultimate limit of a **zettaflops** (10^{21}) flops per second, and the kinds of technologies that might conceivably be necessary to get there. While this series of workshops have included discussions of not just applications, but also device technologies and architectures, the time frame is "at the limits" and not one as "near term" as 2015.

⁴<http://hpcrd.lbl.gov/E3SGS/main.html>

⁵<http://www.zettaflops.org/>

3.7 Prior Relevant Studies

For reference, during the time from 1994 through the present, a long series of studies have continued to build consensus into the importance and justification for advanced computing for all classes of systems from embedded to supercomputing. The sections below attempt to summarize the key findings and recommendations that are most relevant to this study. Since the focus of this study is mostly technical, findings and recommendations discussed below will be largely those with a technical bent: investment and business environment-relevant factors are left to the original reports and largely not covered here.

3.7.1 1999 PITAC Report to the President

The February 1999 *PITAC Report to the President - Information Technology Research: Investing in Our Future*[75] was charged with developing “future directions for Federal support of research and development for information technology.”

The major findings were that Federal information technology R&D investment was inadequate and too heavily focused on near-term problems. In particular relevance to this report, the study found that high-end computing is essential to science and engineering research, it is an enabling element of the United States national security program, that new applications of high-end computing are ripe for exploration, that US suppliers of high-end systems suffer from difficult market pressures, and that innovations are required in high-end systems and application-development software, algorithms, programming methods, component technologies, and computer architecture.

Some major relevant recommendations were to create a strategic initiative in long-term information technology R&D, to encourage research that is visionary and high-risk, and to fund research into innovative computing technologies and architectures.

3.7.2 2000 DSB Report on DoD Supercomputing Needs

The October 2000 Defense Science Board *Task Force on DoD Supercomputing Needs*[17] was charged with examine changes in supercomputing technology and investigate alternative supercomputing technologies in the areas of distributed networks and multi-processor machines.

The Task Force concluded that there is a significant need for high performance computers that provide extremely fast access to extremely large global memories. Such computers support a crucial national cryptanalysis capability. To be of most use to the affected research community, these supercomputers also must be easy to program.

The key recommendation for the long-term was to invest in research on critical technologies for the long term. Areas such as single-processor architecture and semiconductor technology that are adequately addressed by industry should *not* be the focus of such a program. Areas that should be invested in include architecture of high-performance computer systems, memory and I/O systems, high-bandwidth interconnection technology, system software for high-performance computers, and application software and programming methods for high-performance computers.

3.7.3 2001 Survey of National Security HPC Architectural Requirements

The June 2001 *Survey and Analysis of the National Security High Performance Computing Architectural Requirements*[46] was charged with determining if then-current high performance computers that use commodity microprocessors were adequate for national security applications, and also was there a critical need for traditional vector supercomputers.

One key finding was that based on interviews conducted, commodity PCs were providing useful capability in all of 10 DoD-relevant application areas surveyed except for the cryptanalysis area. Another was that while most big applications had scaled well onto commodity PC HPC systems with MPI, several had not, especially those that must access global memory in an irregular and unpredictable fashion.

A summary of the recommendations was to encourage significant research into the use of OpenMP on shared-memory systems, and establish a multifaceted R&D program to improve the productivity of high performance computing for national security applications.

3.7.4 2001 DoD R&D Agenda For High Productivity Computing Systems

The June 2001 *White Paper DoD Research and Development Agenda For High Productivity Computing Systems*[40] was charged with outlining an R&D plan for the HPCS program to revitalize high-end computer industry, providing options for high-end computing systems for the national security community and, developing improved software tools for a wide range of computer architectures. The key findings were that:

- The increasing imbalance among processor speed, communications performance, power consumption, and heat removal results in high-end systems that are chronically inefficient for large-scale applications.
- There exists a critical need for improved software tools, standards, and methodologies for effective utilization of multiprocessor computers. As multi-processor systems become pervasive throughout the DoD, such tools will reduce software development and maintenance - a major cost driver for many Defense system acquisitions.
- Near-elimination of R&D funding for high-end hardware architectures has resulted in a dramatic decrease in academic interest, new ideas, and people required to build the next generation high-end computing systems.

The recommendations were that the attention of academia and industry needed to be drawn to high bandwidth/low latency hierarchical memory systems using advanced technologies, develop highly scalable systems that balance the performance of processors, memory systems, interconnects, system software, and programming environments, and address the system brittleness and susceptibility of large complex computing systems.

3.7.5 2002 HPC for the National Security Community

The July 2002 *Report on High Performance Computing for the National Security Community*[41] was charged with supporting the Secretary of Defense in submitting a development and acquisition plan for a comprehensive, long-range, integrated, high-end computing program to Congress.

The key finding was that the mix of R&D and engineering programs lacked balance and coordination and was far below the critical mass required to sustain a robust technology/industrial base in high-end supercomputing, with requirements identified as critical by the national security community, especially improved memory subsystem performance and more productive programming environments, not being addressed. Another relevant finding was that then current communication interfaces for moving data on and off chips throttled performance.

The key recommendation was to restore the level and range of effort for applied research in fundamental HEC concepts, and apply it nearly evenly across seven general research areas: systems architectures; memory subsystems; parallel languages and programmer tools; packaging/power/thermal management; interconnects and switches; storage and input/output; and novel computational technologies (exclusive of quantum computing).

3.7.6 2003 Jason Study on Requirements for ASCI

The October 2003 Jason study on *Requirements for ASCI*[126] was charged with identifying the distinct requirements of NNSA's stockpile stewardship program in relation to the hardware procurement strategy of the ASCI program.

The two most relevant findings were that a factor of two oversubscription in ASCI Capacity systems was projected to potentially worsen in the foreseeable future, and that future calculations were estimated to take 125X in memory and 500X computations per zone to handle opacity calculations, 40X memory for reactive flow kinetics, and 4X in memory and performance for Sn transport.

Besides increasing ASCI's capability machines, this report also recommended continuing and expanding investments in computational science investigations directed toward improving the delivered performance of algorithms relevant to ASCI.

3.7.7 2003 Roadmap for the Revitalization of High-End Computing

The June 2003 *Workshop on The Roadmap for the Revitalization of High-End Computing*[119] (**HECRTF**) was charged with developing a five-year plan to guide future federal investments in high-end computing.

The overall finding was that short-term strategies and one-time crash programs were unlikely to develop the technology pipelines and new approaches required to realize the Petascale computing systems needed by a range of scientific, defense, and national security applications.

Specific findings and recommendations covered several areas. In enabling technologies, key areas needing work included the management of power and improvements in interconnection performance, the bandwidth and latency among chips, boards, and chassis, new device technologies and 3D integration and packaging concepts, a long-term research agenda in superconducting technologies, spintronics, photonic switching, and molecular electronics, and system demonstrations of new software approaches.

Specific findings and recommendations in architecture were split between **COTS**-based (Custom Off the Shelf) and **Custom**. For the former the emphasis was on exploiting scarce memory bandwidth by new computational structures, and increasing memory bandwidth across the board. For the latter, expected performance advantages of custom features were forecast of between 10X and, programmability advantage of twofold to fourfold. Beyond a relatively near-term period, the report concluded that continued growth in system performance will be derived primarily through brute force scale, advances in custom computer architecture, and incorporation of exotic technologies, and that a steady stream of prototypes were needed to determine viability.

In terms of runtime and operating systems, the study found that the then-research community was almost entirely focused on delivering capability via commodity-leveraging clusters, that for HEC systems the two should merge and need to incorporate much more dynamic performance feedback, and that the lack of large-scale testbeds was limiting such research. In addition, the study concluded that as system sizes increased to 100,000 nodes and beyond, novel scalable and high-performance solutions would be needed to manage faults and maintain both application and system operation.

In terms of programming environments and tools, the study concluded that the most pressing scientific challenges will require application solutions that are multidisciplinary and multiscale, requiring an interdisciplinary team of scientists and software specialists to design, manage, and maintain them, with a dramatic increase in investment to improve the quality, availability, and usability of the software tools that are used throughout an application's life cycle.

3.7.8 2004 Getting Up to Speed: The Future of Supercomputing

The November 2004 National Academy report *Getting Up to Speed The Future of Supercomputing*[54] summarizes an extensive two-year study whose charge was to examine the characteristics of relevant systems and architecture research in government, industry, and academia, identify key elements of context, examine the changing nature of problems demanding supercomputing and the implications for systems design, and outline the role of national security in the supercomputer market and the long-term federal interest in supercomputing.

The key findings were that the peak performance of supercomputers has increased rapidly in the last decades, but equivalent growth in sustained performance and productivity had lagged. Also, the applications described in Section 3.7.3 were still relevant, but that about a dozen additional application areas were identified, with performance needs often 1000X or more would be needed - just over the next five years. Finally, while commodity clusters satisfied the needs of many supercomputer users, important applications needed better main memory bandwidth and latency hiding that are available only in custom supercomputers, and most users would benefit from the simpler programming model that can be supported well on custom systems.

The key recommendations all dealt with the government's necessary role in fostering the development of relevant new technologies, and in ensuring that there are multiple strong domestic suppliers of both hardware and software.

3.7.9 2005 Revitalizing Computer Architecture Research

A 2005 CRA Conference on *Grand Challenges in Computer Architecture*[72] was charged with determining how changes in technology below 65 nm are likely to change computer architectures, and to identify what problems are likely to become most challenging, and what avenues of computer architecture research are likely to be of most value. To quote the report: "Separating computing and communication is no longer useful; differentiating between embedded and mainstream computing is no longer meaningful. Extreme mobility and highly compact form factors will likely dominate. A distributed peer-to-peer paradigm may replace the client-server model. New applications for recognition, mining, synthesis, and entertainment could be the dominant workloads."

The findings of the workshop was that there were four specific challenges whose solutions would potentially have a huge impact on computing in 2020:

1. A "featherweight supercomputer" that could provide 1 teraops per watt in an aggressive 3D package suitable for a wide range of systems from embedded to energy efficient data centers.
2. "Popular parallel programming models" that will allow significant numbers of programmers to work with multi-core and manycore systems.
3. "Systems that you can count on" that provide self-healing and trustworthy hardware and software.

4. “New models of computation” that are not von Neumann in nature, and may better leverage the properties of emerging technologies, especially non-silicon, in better ways for new and emerging classes of applications.

3.7.10 2006 DSB Task Force on Defense Critical Technologies

The March 2006 the *Report on Joint U.S. Defense Science Board and UK Defence Scientific Advisory Council Task Force on Defense Critical Technologies*[18] was charged with examine five transformational technology areas that are critical to the defense needs of the US and UK, including Advanced Command Environments, Persistent Surveillance, Power sources for small distributed sensor networks, High Performance Computing, and Defence Critical Electronic Components.

They key finding on the HPC side were that there are a number of applications that cannot be solved with sufficient speed or precision today, that the high performance needs of national security will not be satisfied by systems designed for broader commercial market, and that two new memory-intensive applications are becoming especially important: knowledge discovery and integration and image and video processing.

The key recommendations on the HPC side were to make HPCS a recurring program, and invest in technologies especially for knowledge discovery including new very large memory-centric systems, programming tools, system software, improved knowledge discovery algorithms that can run unattended, new inference engines, support for rapid, high productivity programming, appropriate metrics, and open test beds that permit research community to explore and evaluate without revealing national securing information.

The recommendations on knowledge discovery and video processing were buttressed by the findings and recommendations on **Persistent Surveillance**. In particular a recommendation was to establish a US persistent surveillance effort.

3.7.11 2006 The Landscape of Parallel Computing Research

The report *The Landscape of Parallel Computing Research: A View From Berkeley*[11] summarized a two-year effort by researchers at the University of California at Berkeley to discuss the effects of recent emergence of multi-core microprocessors on highly parallel computing, from applications and programming models to hardware and evaluation.

Their major finding was that multi-core was unlikely to be the ideal answer to achieving enhanced performance, and that a new solution for parallel hardware and software is desperately needed. More detailed findings included that

- It is now memory and power that are the major walls.
- As chips drop below 65 nm feature sizes, they will have higher soft and hard error rates that must be accounted for in system design.
- Both instruction level parallelism and clock rates have reached points of diminishing returns, and conventional uniprocessors will no longer improve in performance by 2X every 18 months.
- Increasing explicit parallelism will be the primary method of improving processor performance.
- While embedded and server computing have historically evolved along separate paths, increasingly parallel hardware brings them together both architecturally and in terms of programming models.

A key output of the report was the identification of 13 benchmark dwarves that together can delineate application requirements in a way that allows insight into hardware requirements. In terms of hardware projections, the report suggests that technology will allow upwards of a 1000 simple and perhaps heterogeneous cores on a die (**manycore**), but that separating DRAM from CPUs as is done today needs to be seriously revisited. Further, interconnection networks will become of increasing importance, both on and off chip, with coherency and synchronization of growing concern. In addition, more attention must be paid to both dependability and performance and power monitoring to provide for **autotuners**, which are software systems that automatically adapt to performance characteristics of hardware, often by searching over a large space of optimized versions. New models of programming will also be needed for such systems.

Chapter 4

Computing as We Know It

This chapter discusses the structure and inter-relationship of computing systems today, including architecture, programming models, and resulting properties of applications. It also includes some historical data on microprocessor chips, on the leading supercomputer systems in the world over the last 20 years, and on web servers which make up many of the departmental sized systems of today.

4.1 Today's Architectures and Execution Models

This section should overview today's deep memory hierarchy-based system architectures using hot multi-core processing chips, dense DRAM main memory, and spinning, disk-based, secondary memory.

4.1.1 Today's Microarchitectural Trends

Contemporary microprocessor architectures are undergoing a transition to simplicity and parallelism that is driven by three trends. First, the instruction-level parallelism and deep pipelining (resulting in high clock rates) that accounted for much of the growth of single-processor computing performance in the 1990s has been *mined out*. Today, the only way to increase performance beyond that achieved by improved device speed is to use explicit parallelism.

Second, the constant field scaling that has been used to give "cubic"¹ energy scaling for several decades has come to an end because threshold voltage cannot be reduced further without prohibitive subthreshold leakage current. As a result, power is *the* scarce resource in the design of a modern processor, and many aggressive superscalar techniques that burn considerable power with only modest returns on performance have been abandoned. Newer processors are in many cases simpler than their predecessors to give better performance per unit power.

Finally, the increase in main memory latency and decrease in main memory bandwidth relative to processor cycle time and execution rate continues. This trend makes memory bandwidth and latency the performance-limiting factor for many applications, and often results in sustained application performance that is only a few percent of peak performance. Many mainstream processors now adopt aggressive latency hiding techniques, such as out-of-order execution and explicit multi-threading, to try to tolerate the slow path to memory. This, however, has an element of self-defeatism, since they often results in worse than linear increase in the complexity of some ba-

¹The energy per device shrinking as the 3rd power of the reduction in the device's basic feature size

sic structure (such as comparators associated with a load-store buffer)to contributes a worse than linear growth in power for a less than linear increase in performance.

4.1.1.1 Conventional Microprocessors

Conventional high-end microprocessors aim for high single-thread performance using speculative superscalar designs. For example, the AMD K8 Core microarchitecture includes a 12 stage pipeline that can execute up to 3 regular instructions per cycle. The complex out-of-order instruction issue, retirement logic, and consumes considerable power and die area; only about 10% of the core die area (not including L2 caches) is devoted to arithmetic units.

At the other end of the spectrum, Sun's Niagara 2 chip[110] includes 8 dual-issue cores, each of which supports up to 8-way multi-threading. The cores are simpler, with in-order execution, sacrificing single-thread performance for the benefit of greater thread-level parallelism. These simpler processors are more efficient, but still consume considerable area and energy on instruction supply, data supply, and overhead functions.

To amortize the high overheads of modern processors over more operations, many processors include a *short vector* instruction set extension such as Intel's SSE. These extensions allow two or four floating point operations to be issued per cycle drawing their operands from a wide (128b) register file. Emerging processors such as Intel's Larrabee take this direction one step further by adding a wider vector processor.

4.1.1.2 Graphics Processors

Performance in contemporary **graphics processing units (GPUs)** has increased much more rapidly than conventional processors, in part because of the ease with which these processors can exploit parallelism in their main application area. Modern GPUs incorporate an array of programmable processors to support the programmable shaders found in graphics APIs. For example, the Nvidia GForce 8800 includes an array of 128 processors each of which can execute one single-precision floating-point operation each cycle.

The programmability and high performance and efficiency of modern GPUs has made them an attractive target for scientific and other non-graphics applications. Programming systems such as Brook-GPU [22] and Nvidia's CUDA [111] have evolved to support general purpose applications on these platforms. Emerging products such as AMD's *fusion* processor are expected to integrate a GPU with a conventional processor to support general-purpose applications.

4.1.1.3 Multi-core Microprocessors

In the early 2000s' the confluence of limitations on per chip power dissipation and the flattening of our ability to trade more transistors for higher ILP led to a stagnation in single-core single-thread performance, and a switch in chip level microarchitectures to die with multiple separate processing cores on them. This switch has occurred across the board from general purpose microprocessors to DSPs, GPUs, routers, and game processors, and has taken on the generic names of **multi-core**, **manycore**, or **chip-level multi-processing (CMP)**.

The rise of multiple cores on a single die has also introduced a new factor in a die's microarchitecture: the interconnect fabric among the cores and between the cores and any external memory. There are currently at least three different classes[86] of such die-level interconnect patterns, Figure 4.1:

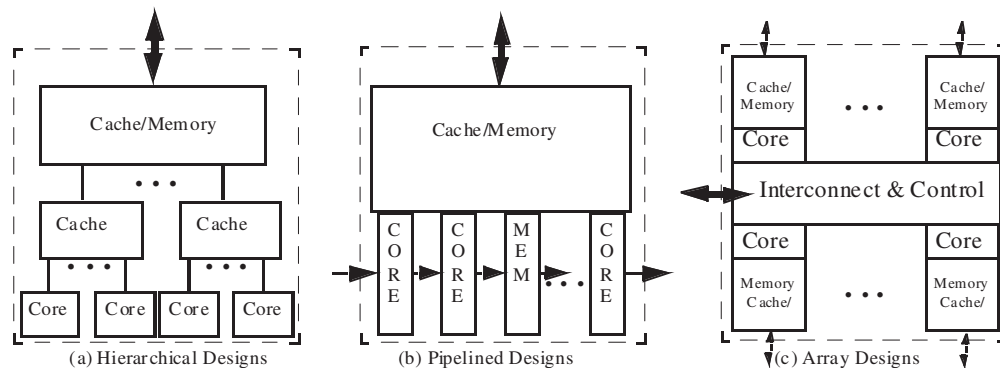


Figure 4.1: Three classes of multi-core die microarchitectures.

- **Hierarchical:** where cores share multi-level caches in a tree-like fashion, with one or more interfaces to external memory at the top of the tree. Most general purpose microprocessors fall into this category.
- **Pipelined:** where off-chip inputs enter one core, which then may communicate with the “next” core, and so on in a pipelined fashion, and where the “last” core then sends processed data off-chip. On-chip memory may be either as “stage” in its own right, associated with a unique core, or part of a memory hierarchy as above. Many router and graphics processors follow this microarchitecture, such as the Xelerator X10q[25], which contains 200 separate cores, all arranged in a logically linear pipeline.
- **Array:** where each core has at least some local memory or cache, may or may not share a common memory, and may communicate with other cores either with some sort of X-Y routing or through a global crossbar. Examples of designs for 2 or 3D X-Y interconnect include both EXECUBE[85][144] and Intel’s **Teraflops Research Chip**[149]; examples of crossbar interconnect include Sun’s Niagara 2 chip[110] and IBM’s **Cyclops**[8]; examples of reconfigurable tiles include **RAW**[146].

4.1.2 Today’s Memory Systems

Today’s general purpose computer systems incorporate one or more **chip multiprocessors (CMPs)** along with external DRAM memory and peripherals. Each processor chip typically falls into the hierarchical arrangement as discussed above, and combines a number of processors (currently ranging from 2-24) along with the lower levels of the memory hierarchy. Each processor typically has its own **level-1 (L1)** data and instruction caches. Some CMPs also have private L2 caches, while others share a banked L2 cache (interleaved by cache block) across the processors on the chip. Some CMPs, such as the Itanium-2TM, have large (24MB) on-chip shared level-3 caches. Some manufacturers (Sun and Azul) provide some support for transactional memory functionality.

The on-chip processors and L2 cache banks are connected by an on-chip interconnection network. In the small configurations found today, this network is typically either a bus, a crossbar, or a ring. In the larger configurations expected in the near future, more sophisticated on-chip networks will be required.

The main memory system is composed of conventional (DDR2) DRAM chips packaged on small daughter cards often called variants of **DIMMs (Dual Inline Memory Modules)**. A

small number of data bits leave the DIMM from each chip, usually 4, 8 or 9 bits. In most cases, the data outputs of these chips connect directly to the processor chip, while in other cases they connect to a **north bridge** chip that is connected to the processor via its **front-side bus**. In both cases a **memory controller** is present either on the CPU, the north bridge, or as a separate chip, and whose purpose is to manage the timing of accesses to the DRAM chips for both accesses and for the refresh operations that are needed to keep their contents valid.

In today's cache-based memory systems, the processor reads and writes in units of words, but all transfers above the level of the L1 caches take place in units of cache lines which range from 32B to 128B. When there is a miss in the L1 cache, an entire line is transferred from L2 or from main memory. Similarly when a dirty line is evicted from a cache, the entire line is written back. Particularly with the larger granularity cache lines, such a line oriented memory system reduces the efficiency of random fine-grained accesses to memory.

Pin bandwidth of modern DRAMs are about 1Gb/s using the **Stub Series Terminated Logic (SSTL)** signaling standard. Modern commodity processors use interfaces ranging from 8B to 32B wide to provide from 8 to 32GB/s of main memory bandwidth. This per pin rate is relatively low because of the technology used on the DRAMs, and because of power consumption in the interface circuits. In addition, typical DIMM4 may be paralleled up to 4 ways to increase the memory capacity without expanding the pin count on the microprocessor. This increases the capacitance on the data pins, thus making higher rate signalling more difficult.

One new technique to increase the data rate from a multi-card memory system is called **Fully Buffered DIMM (FB-DIMM)**, and uses a higher speed serialized point-to-point data and control signalling protocol from the microprocessor side to an **Advanced Memory Buffer (AMB)** chip on each FB-DIMM memory card. This AMB then talks to the on-card DRAM parts to convert from their parallel data to the protocol's serialized format. The higher speed of the protocol is possible because each link is uni-directional and point-to-point, and multiple FB-DIMM cards are arranged in a daisy-chain fashion with separate links going out and coming back. Additional bandwidth is gained because this arrangement allows accesses to be pipelined both in terms of outgoing commands and returning data.

A typical contemporary microprocessor has latencies of 1 cycle, 20 cycles, and 200 cycles to the L1 cache, L2 cache, and external DRAM respectively. Memory systems such as FB-DIMM actually increase such latencies substantially because of the pipelined nature of the card-to-card protocol. These latencies make the penalty for a cache miss very high and motivate the use of latency-hiding mechanisms.

4.1.3 Unconventional Architectures

Overcoming the memory wall has been a major emphasis of alternative architectures that have begun to crop up, especially with the emergence of multi-core architectures that place extra pressure on the memory system. Multi-threading has become more acceptable, with dual threads present in many main-line microprocessors, and some chips such as Sun's Niagara 2TM chip[110] support even more. Cray's MTATM[132] and XMTTM[44] processors were designed from the outset as multi-threaded machines, with in excess of 100 threads per core supported.

Stream processors such as the IBM Cell[57] and SPI Storm-1TM are an emerging class of architecture that has arisen to address the limits of memory latency and bandwidth. These processors have explicitly managed on-chip memories in place of (or in addition to) the conventional L2 cache. Managing the contents of these memories explicitly (vs. reactively as with a cache) enables a number of aggressive compiler optimizations to hide latency and improve the utilization of scarce bandwidth. Explicit management of the memory hierarchy, however, places additional burdens on

the application programmer and compiler writer.

Processing-In-Memory (PIM) chips reduce the latency of a memory access, and increase the effective bandwidth by using more of the bits that are read from a dense, usually DRAM, memory array at each access by placing one or more processors on the chip with the memory. A variety of different processor architectures and interconnect patterns have been used, ranging from arrangements of more or less conventional cores on **EXECUBE**[85], **Cyclops**[8] and Intel's **Teraflops Research Chip**[149], "memory chips" with embedded processing such as **DIVA**[60] and **YUKON**[83], and **VIRAM**[88] - where multiple vector units are integrated onto a DRAM.

In addition, a new class of **reconfigurable multi-core architectures** are entering evaluation where a combination of large and small grained concurrency is exploited by tiles of cores that may be reconfigured on an algorithm by algorithm basis. These include **TRIPS**[123] and **RAW**[146].

4.1.4 Data Center/Supercomputing Systems

While no clear definition exists, **supercomputing systems** are commonly defined as the most computationally powerful machines available at any time. Since the mid 1990s, all supercomputers have been designed to be highly scalable, and thus a given system design may be scaled down or up over a broad range. A typical supercomputer today may sell in configurations costing O(\$1M) to O(\$100M).

4.1.4.1 Data Center Architectures

Supercomputing systems fall into two broad classes: clusters of conventional compute nodes, and custom systems. Custom systems are machines where the individual compute nodes are designed specifically for use in a large scientific computing system. Motivated by demanding scientific applications, these nodes typically have much higher memory bandwidth than conventional processors, very high peak and sustained arithmetic rates, and some form of latency hiding mechanism — often vectors as in the Earth Simulator. These custom processing nodes are connected by an interconnection network that provides access to a global shared memory across the machine. Global memory bandwidth is usually a substantial fraction of local memory bandwidth (e.g. 10%) and global memory latency is usually only a small multiple of local memory latency (e.g. 4x).

Clusters of conventional nodes are, as the name suggests, conventional compute nodes connected by an interconnection network to form a cluster. While most such nodes may use "mass-market" chips, there are significant examples of specially designed chips that leverage prior processor designs (such as the Blue Gene/L systems) Further, most such compute nodes are themselves SMPs, either as collections of microprocessors, or increasingly multi-core processors, or both. These in fact make up the majority of systems on the Top 500.

The individual nodes typically have separate address spaces and have cache-based memory systems with limited DRAM bandwidth and large cache-line granularity. The nodes are connected together by an interconnection network that can range from conventional 1G or 10G ethernet to a cluster-specific interconnect such as **Infiniband**², **Myrinet**³, **Quadrics**⁴, or even custom high bandwidth interconnect such as the SeaStar[21] used in the Red Storm and derivatives in the Cray XT line. Global bandwidth is usually a small fraction of local bandwidth and latencies are often several microseconds (with much of the latency in software protocol overhead).

²Infiniband standard can be found at <http://www.infinibandta.org/>

³Myrinet standards can be found at <http://www.myri.com/open-specs/>

⁴<http://www.quadrics.com>

Conversion Step	Efficiency	Delivered	Dissipated	Delivered	Dissipated
AC In		1.00W		2.06W	
Uninterruptible Power Supply (UPS)	88%	0.88W	0.12W	1.81W	0.25W
Power Distribution Unit (PDU)	93%	0.82W	0.06W	1.69W	0.13W
In Rack Power Supply Unit (PSU)	79%	0.65W	0.17W	1.33W	0.35W
On Board Voltage Regulator (VR)	75%	0.49W	0.16W	1.00W	0.33W
Target Logic		0.49W	0.49W	1.00W	1.00W

Table 4.1: Power distribution losses in a typical data center.

Another distinguishing characteristic for the interconnect is support addressing extensions that provide the abstraction of a globally addressable main memory (as in the **pGAS** model). Several interconnect protocols such as Infiniband and SeaStar provide such capabilities in the protocol, but using them effectively requires support for large address spaces that, if not present in the native microprocessor, in turn may require specialized network interface co-processors.

4.1.4.2 Data Center Power

Perhaps the single biggest impact of scale is on system power. Table 4.1 summarizes the different steps in converting wall power to power at the logic for a typical data center today as projected by an Intel study[7]. For each step there is:

- an “Efficiency” number that indicates what fraction of the power delivered at the step’s input is passed to the output,
- a “Delivered” number that indicated how much of the power (in watts) is passed on to the output if the power on the line above is presented as input,
- and a “Dissipated” power (again in watts) that is lost by the process.

There are two sets of the Delivered and Dissipated numbers: one where the net input power is “1,” and one where the power delivered to the logic is “1.” The net efficiency of this whole process is about 48%, that is for each watt taken from the power grid, only 0.48 watts are made available to the logic, and 0.52 watts are dissipated as heat in the power conversion process.

A similar white paper by Dell[38] gives slightly better numbers for this power conversion process, on the order of about 60%, that is for each watt taken from the power grid, 0.6 watts are made available to the logic, and 0.4 watts are dissipated as heat in the power conversion process.

This same source also bounds the cost of the **HVAC** (heating, ventilating, and air conditioning) equipment needed to remove the heat dissipated by the data center at about 31% of the total data center power draw. This represents an energy tax of about 52% on every watt dissipated in either the logic itself or in the power conditioning steps described above.

Since the focus on this report is on analyzing the power dissipation in a potential Exascale data center, these numbers roll up to an estimate that for every watt dissipated by the electronics, somewhere between 0.7 and 1.06 watts are lost to the power conditioning, and between 0.88 and 1.07 watts are lost to cooling. Thus the ratio between the power that may be dissipated in processing logic and the wattage that must be pulled from the grid is a ratio of between 2.58 and 3.13 to 1.

As a reference point, Microsoft is reported⁵ to be building a data center near Chicago with an initial power source of up to 40 MW. Using the above numbers, 40 MW would be enough for about 13 to 15 MW of computing - about 2/3 of the Exascale data center goal number.

4.1.4.2.1 Mitigation Mitigation strategies are often employed to help overcome the various losses in the electricity delivery system. For example:

- **UPS:** At these power levels, many data centers choose to forgo UPS protection for the compute nodes and only use the UPS to enable graceful shutdown of the disk subsystem (a small fraction of the overall computer power dissipation). In the case where the UPS is used for the entire data center, DC power is an increasingly studied option for reducing losses. The primary source of losses within a UPS are the AC-DC rectification stage used to charge the batteries, and then the DC-to-AC inversion stage used to pull power back off of the batteries at the other end (which will again be rectified by the compute node power supplies). DC power cuts out an entire layer of redundant DC-AC-DC conversion. Below 600VDC, the building electrical codes and dielectric strengths of electrical conduits and UL standards are sufficient, so most of the existing AC electrical infrastructure can be used without modification. Care must be taken to control arc-flashes in the power supply connectors. In addition, the circuit design modifications required to existing server power supplies are minimal. Using DC power, one can achieve ~95% efficiency for the UPS.
- **PDU:** Using high-voltage (480VAC 3-phase) distribution within the data center can reduce a layer of conversion losses from PDUs in the data center and reduce distribution losses due to the resistivity of the copper wiring (or allow savings in electrical cable costs via smaller diameter power conduits).
- **PSUs:** The Intel and Dell studies highlight the power efficiency typically derived from low-end 1U commodity server power supplies (pizza-box servers). Using one large, and well-engineered power supply at the base of the rack and distributing DC to the components, power conversion efficiencies exceeding 92% can be achieved. Power supplies achieve their best efficiency at 90% load. When redundant power supplies are used, or the power supplies are designed for higher power utilization than is drawn by the node, the power efficiency drops dramatically. In order to stay at the upper-end of the power conversion efficiency curve one must employ circuit-switching for redundant feeds, or N+1 redundancy (using multiple power supply sub-units) rather than using the typical 2N redundancy used in modern commodity servers. Also, the power supply should be tailored to the actual power consumption of the system components rather than over-designing to support more than one system configuration.

The overall lesson for controlling power conversion losses for data centers is to consider the data center as a whole rather than treating each component as a separate entity. The data-center is the computer and power supply distribution chain must be engineered from end-to-end starting with the front-door of the building. Using this engineering approach can reduce power-distribution losses to ~10% of the overall data center power budget.

4.1.4.3 Other Data Center Challenges

Regardless of the class of system, supercomputers face several unique challenges related to scale. These include reliability, administration, packaging, cooling, system software scaling and performance scaling. A large supercomputer may contain over 100 racks, each containing over 100

⁵<http://www.datacenterknowledge.com/archives/2007/Nov/05/microsoft-plans-500m-illinois-data-center.html>

processor chips, each connected to tens of memory chips. Reliability is thus a major concern, and component vendors targeting the much larger market of much smaller-scale machines may not be motivated to drive component reliability to where it needs to be for reliable operation at large scales.

Density is important in supercomputing systems, both because machine room floor may be at a premium, and also because higher density reduces the average length of interconnect cables, which reduces cost and increases achievable signaling rates. High density in turn creates power and cooling challenges. Current systems can dissipate several tens of kilowatts per rack, and several megawatts of total power.

Careful attention must be paid to all aspects of system administration and operation. System booting, job launch, processor scheduling, error handling, file systems, networking and miscellaneous system services must all scale to well beyond what is needed for the desktop and server markets. Removing sources of contention and system jitter are key to effective scaling. Finally, the applications themselves must be scaled to tens of thousands of threads today, and likely millions of threads within the next five to ten years.

4.1.5 Departmental Systems

Today's departmental systems fall into two main categories. There are symmetric multiprocessors (SMPs) that have one large, coherent main memory shared by a few 64-bit processors such as AMD's OpteronsTM, IBM's Power 6TM and z series, Intel's XeonTM and ItaniumTM, and Sun's UltraSparcTM T-series (also known as Niagara). The main memory is usually large, as is the accompanying file system. It is not uncommon to see systems with a terabyte of memory and only a handful of CPUs. These systems are often used in mission critical applications such as on-line transaction processing. Thus they offer their users higher availability than systems based on PC components. Most importantly, they provide a graceful transition path to concurrent execution for commercial applications whose software architecture goes back earlier than the mid-1990s, and hence do not support a partitioned address space.

The second category of departmental systems is composed of clusters of PC boxes (often called **Beowulf** systems), and at first glance resemble smaller versions of the supercomputers discussed above. The main difference is that the departmental clusters generally do not include relatively expensive components like the custom high-speed interconnects found in the supercomputers. Such clusters also tend to run the Linux operating system, and are generally used to host a job mix consisting of large ensembles of independent jobs that each use only a small number of CPUs.

4.1.6 Embedded Systems

Embedded systems is a term that covers a broad range of computers from ubiquitous hand held consumer items like cellular phones to Defense reconnaissance satellites. In fact, the number of CPUs manufactured for embedded applications dwarfs those for desktops and servers. Embedded systems tend to be optimized for lower power consumption (AA battery lifetime is a key metric) which in turn allows for higher packaging density. Flash memories are used to provide a modest volume of persistent memory, and interconnection is often provided via ad hoc wireless networks.

Embedded systems usually run small, real-time operating systems. These are more sophisticated than the micro-kernels on some supercomputers, but do not support the broad range of features expected by desktop or server applications. Today's embedded applications are generally written in C. Historically, they were different than the applications seen in desktops and servers, but with the new generation of hand-held Web interfaces (e.g. Apple's iPhoneTM), the end user application

space may tend to become indistinguishable.

4.1.7 Summary of the State of the Art

The state of the art in computer architecture today is dominated by a 50 year reliance on the von Neumann model where computing systems are physically and logically split between memory and CPUs. Memory is “dumb,” with the only functionality being able to correlate an “address” with a particular physical set of storage bits. The dominant execution model for CPUs is that of a sequential thread, where programs are divided into lists of instructions that are logically executed one at a time, until completion, and each operation applies to only a relatively small amount of data. Higher performance architectures invest a great deal of complexity in trying to place more instructions in play concurrently, without the logical model of sequential execution being violated, especially when exceptions may occur.

Two technological developments have impacted this model of computer architecture. First is the **memory wall**, where the performance of memory, in terms of access time and data bandwidth, has not kept up with the improvement in clock rates for the CPU designs. This has led to multi-level caches and deep memory hierarchies to try to keep most of the most relevant data for a program close to the processing logic. This in turn has given rise to significantly increased complexity when multiple CPUs attempt to share the same memory, and collaborate by explicitly executing separate parts of a program in parallel.

The second technological development that has impacted architecture is the rise of power as a first class constraint (the **power wall**), and the concomitant flattening of CPU clock rates. When coupled with the memory wall, performance enhancement through increasing clock rates is no longer viable for the mainstream market. The alternative is to utilize the still growing number of transistors on a die by simple replication of conventional CPU cores, each still executing according to the von Neumann model. Chips have already been produced with a 100+ cores, and there is every expectation that the potential exists for literally thousands of cores to occupy a die. The memory wall, however, is still with us, with a vengeance. Regardless of processor chip architectures, more cores usually means more demand for memory access, which drives a requirement for more memory bandwidth. Unfortunately, current technology sees a flattening of the number of off-chip contacts available to a processor chip architect, and the rate at which these contacts can be driven is tempered by power concerns and the technology used to make memory chips - one chosen for density first, at low fabrication cost. High performance interfaces in such an environment are difficult.

As Section 4.1.3 describes, there are attempts today to modify this von Neumann model by blurring in various ways the distinction between memory “over here” and processing logic “over there.” To date, however, the penetration into the commercially viable sector has been nearly nil.

4.2 Today’s Operating Environments

An **operating environment** is the environment in which users run programs. This can range from a graphical user interface, through a command line interface, to a simple loader, run-time scheduler, memory manager, and API through which application programs can interact with system resources. Conventional software computing environments supporting high performance computing platforms combine commercial or open source node local operating system software and compilers with additional middle-ware developed to coordinate cooperative computing among the myriad nodes comprising the total system (or some major partition thereof). For technical computing, local node operating systems are almost exclusively based on variants of the Unix operating

system introduced initially in the mid to late 1970s. While a number of successful commercial Unix OS are provided by vendors, most notably IBM (AIX), Sun (Solaris), and HP (HPUX), the dominant operating system in today's production level high performance computing machines is the Open Source Unix variant, Linux. In addition, a recent offering by Microsoft is providing a Windows based HPC environment. On top of this largely Unix based node local foundation is additional "middleware" for user access, global system administration, job management, distributed programming, and scheduling.

In addition to these heavy-weight operating systems is a class of lightweight kernels. These have been employed on some of the largest supercomputers including the IBM BlueGene/L system (its own microkernel) and the Cray XT series (Compute Node Linux - a trimmed down Linux).

Further, most system actually employ nodes with a mix of operating environments, with service or I/O nodes typically running a more complete offering to provide more complete functionality.

Finally, **Catamount** is a tailored HPC run-time that has run on several HPC systems, such as Red Storm and Cray's XT3 and XT4. Each is discussed briefly below.

4.2.1 Unix

Several key attributes of the Unix kernel make the system highly customizable and portable, especially as realized in Linux. At the core of the various Linux distributions, the modular monolithic kernel is a contributing factor to the success of Linux in HPC. The Linux kernel is layered into a number of distinct subsystems and the core ties these various modules together into the kernel.

The main subsystems of the Linux kernel are:

- **System Call Interface (SCI):** This is an architecture dependent layer that provides the capability of translating user space function calls to kernel level calls.
- **Process management:** This provides ability for active threads to share CPU resources based on a predefined scheduling algorithm. The Linux kernel implements the **O(1) Scheduler** that supports symmetric multiprocessing.
- **Memory Management:** Hardware memory is commonly virtualised into 4KB chunks known as **pages**, with the kernel providing the services to resolve these mapping. Linux provides management structures such as the **slab allocator** to manage full, partially used and empty pages.
- **File system management:** Linux uses the **Virtual File System interface (VFS)** (a common high level abstraction supported by various file systems) to provide an access layer between the kernel SCI and VFS-supported file systems.
- **Network subsystem:** The kernel utilizes the sockets layer to translate messages into the TCP(/IP) and UDP packets in a standardized manner to manage connections and move data between communicating endpoints.
- **Device Drivers:** These are key codes that help the kernel to utilize various devices, and specify protocol description so that the kernel (SCI) can easily access devices with low overheads. The Linux kernel supports dynamic addition and removal of software components (dynamically loadable kernel modules).

4.2.2 Windows NT Kernel

Windows Server 2003 has a hybrid kernel (also known as **macrokernel**) with several emulation subsystems run in the user space rather than in the kernel space.

The **NT** kernel mode has complete access to system and hardware resources and manages scheduling, thread prioritization, memory and hardware interaction. The kernel mode comprises of executive services, the microkernel, kernel drivers, and the hardware abstraction layer.

- **Executive:** The executive service layer acts as an interface between the user mode application calls and the core kernel services. This layer deals with I/O, object management, security and process management, local procedural calls, etc through its own subsystems. The executive service layer is also responsible for cache coherence, and memory management.
- **MicroKernel:** The microkernel sits between the executive services and the hardware abstraction layer. This system is responsible for multiprocessor synchronization, thread and interrupt scheduling, exception handling and initializing device drivers during boot up.
- **Kernel-mode drivers:** This layer enables the operating system to access kernel-level device drivers to interact with hardware devices.
- **Hardware Abstraction Layer (HAL):** The HAL provides a uniform access interface, so that the kernel can seamlessly access the underlying hardware. The HAL includes hardware specific directives to control the I/O interfaces, interrupt controllers and multiple processors.

4.2.3 Microkernels

A microkernel is a minimal run-time which provides little or no operating system services directly, only the mechanisms needed to implement such services above it.

Compute Node Linux[152] (**CNL**) is an example of one such that has its origins in the SUSE SLES distribution.

The microkernel for BlueGene/L[108] was designed to fit in small memory footprints, and whose major function would be to support scalable, efficient execution of communicating parallel jobs.

Catamount[152] is an independently-developed microkernel based operating system that has two main components: the **quintessential kernel** and a **process control thread**. The process control thread manages physical memory and virtual addresses for a new process, and based on requests from the process control thread the quintessential kernel sets up the virtual addressing structures as required by the hardware. While Catamount supports virtual addressing it does not support virtual memory. The process control thread decides the order of processes to be executed and the kernel is responsible for flushing of caches, setting up of hardware registers and running of the process. In essence the process control thread sets the policies and the quintessential kernel enforces the policies. Catamount uses large (2 MB) page sizes to reduce cache misses and TLB flushes.

- **Quintessential Kernel:** The **Q. Kernel (QK)** sits between the process control thread (**PCT**) and the hardware and performs services based on requests from PCT and user level processes. these services include network requests, interrupt handling and fault handling. if the interrupt or fault request is initiated by the application then the QK turns over the control to PCT to handle those requests. . The QK handles privileged requests made by PCT such as running processes, context switching, virtual address translation and validation.

- **Process Control Thread:** The PCT is a special user-level process with read/write access to the entire memory in user-space and manages all operating system resources such as process loading, job scheduling and memory management. QKs are non communicating entities, while the PCTs allocated to an application can communicate to start, manage and kill jobs.
- **Catamount System Libraries:** While QK and PCT provide the mechanisms to effect parallel computation, Catamount libraries provide applications access to harness their mechanisms. Catamount system libraries comprise of Libc, libcatamount, libsysio and libportals, Applications using these services have to be compiled along with the Catamount system libraries.

4.2.4 Middleware

Additional system software is provided to simplify coordination of the distributed resources of the scalable MPPs and commodity clusters most prevalent today. Tool sets such as **PBS**, the **Maui scheduler**, and **Loadleveler** provide such system wide environments. Management of secondary storage from parallel systems is provided by additional file system software such as the open source **PVFS** and Lustre systems and the IBM proprietary GPFS. Although Lustre is open source, is it supported and maintained in large part by Sun.

4.2.5 Summary of the State of the Art

The vast majority of operating systems on current the Top-500 list are Unix or its derivatives and the majority of these are Linux in various forms. Unix provides a well understood and time tested set of services and interfaces. Systems today are ensembles of compute subsystems with the majority organized as distributed memory collections. The operating environments reflect this division by providing node operating systems with supporting middleware to coordinate their scheduling and allocation to user workloads. Therefore, as the physical system is a collection of multiprocessor or multicore nodes integrated via one or more system interconnect networks, so the logical system is a collection of Unix or Linux domains managing distinct sets of cores integrated by a scheduling layer. A separate but important piece of the operating environment is the mass storage system including the file system. Parallel file systems manage large numbers of disk drives often with multiple controllers each servicing a number of spindles. An emerging trend in operating environments is the use of lightweight kernels to manage some or all of the compute nodes. The original environment on the Cray XT3 was the Catamount lightweight kernel system. The IBM Blue Gene system has mixed mode environments with a lightweight kernel on each of its compute nodes managed by an I/O node hosting a Linux system. The challenge for the immediate future is to develop advanced Unix and Linux versions that can support up to hundreds of cores per shared memory node. But over the longer term, lightweight kernel systems may prove valuable for systems comprising millions of small cores such as the Cell architecture.

4.3 Today's Programming Models

A **programming model** is comprised of a set of languages and libraries that define the programmer's view of a machine. Examples of programming models for parallel machines include traditional sequential languages combined with a message passing layer or a thread library, parallel languages like UPC or HPF, and sequential languages combined with parallelizing compilers. A single machine may support multiple programming models that are implemented through compiler translation and

run-time software layers. In a broad sense, each parallel programming model combines a control model that specifies how the parallelism is created and managed, and a communication model that allows the parallel entities to cooperate somehow by sharing data.

An **execution model** is distinct from the programming model and consists of the physical and abstract objects that comprise the evolving state of the computation. It defines how the physical and abstract objects that actually perform the computation and track its results interact. An execution model ties together all layers of the vertical functionality stack from the application and algorithms by means of the programming language and compiler, the run-time and operating system, and goes down to the system wide and local micro-architecture and logic design. The execution model conceptually binds all elements of a computing system in to a single coordinated cooperating computing corporation. An execution model may consist of a fixed set of thread or process states, each with its own program stack, and a set of message queues and other state information needed to communicate between the threads. A programming model may be very similar to its underlying execution model, although in some cases they can also be quite different. For example, a data parallel languages like HPF provide the illusion of an unbounded set of lightweight processors that execute each statement in lock step, while compilers for distributed memory architectures convert this data parallel code into a smaller set of physical threads that correspond to the number of physical processors; the threads execute independently and communicate with explicit message passing.

The execution model determines how a program executes, whereas the programming model determines how it is expressed.

As computing technology has evolved it has enabled ever more powerful, often increasingly parallel, computer architectures. Innovations in execution models have included vector and SIMD, dataflow and systolic, message passing and multi-threaded, with a diversity of variations of each. Such execution models have differed in the form of parallelism explicitly exploited, while taking advantage of others intrinsic to the micro-architecture structures such as ILP, pipelines, and prefetching. The programming models have similarly evolved, starting with serial languages (with automatic parallelization) on the vector execution model, data parallel languages with SIMD execution, and functional languages on dataflow.

The **communicating sequential processes**[65] (**CSP**) execution model (essentially a formalism of message passing) has dominated much of the last two decades and drives many of the largest systems today including the IBM Blue Gene, Cray XTn, and commodity Linux clusters. Execution models based on threads communicating through shared variables are often used on smaller machines with cache coherent shared memory, and a hybrid model may be used on machines that are built from as clusters of shared memory nodes, especially on the IBM Power architectures with a relatively large number of processor cores per node. Performance experience with the hybrid model is mixed, and in many cases programmers use message passing processes even on individual cores of a shared memory node.

Users select programming models based on a number of factors, including familiarity, productivity, convenience, performance, compatibility with existing code, and portability across current and future machines. Today's Terascale codes are written predominantly in the MPI Message Passing Interface using a traditional sequential language like Fortran, C or C++. Mixed language applications are also quite common, and scripting languages like Python are often used in putting together complex application workflows built from multiple programming models.

Machine architectures may enable newer and (by some criteria) better programming models, but there is a great deal of inertia in existing models that must be overcome to move a significant fraction of the user community. Historically, this has happened when the execution model changes significantly due to underlying architectural changes, and the programming model of choice is either

unsupported or offers reduced performance capabilities. When vector machines were the dominant high end architecture, they were programmed primarily with annotated sequential languages, such as Fortran with loop annotations. When large-scale SIMD machines were popular (e.g. the CM-2 and Maspar), data parallel languages like CMFortran, *Lisp, and C* were the preferred model. These models were quickly overtaken by message passing models as distributed memory machines including low-cost Beowulf clusters became the architecture of choice.

In each case, there were efforts to move the previous model to an execution model that could be supported on the new machines. Attempts to move automatic parallelism from the vector execution model to shared memory threads was not successful in practice, in spite of significant investment and many positive research results on the problem. Motivated by the success of vector compiler annotations, OpenMP was designed to provide a similar programming model for the shared memory execution model and this model is successful, although the annotations are significantly more complex than earlier vector annotations. The HPF language effort was initiated to take a data parallel model to message passing hardware, but performance was not considered adequate for larger scale machines. Similarly, despite many efforts to provide shared memory programming on a message passing execution model, this is not used in practice today. Partitioned Global Address Space languages like UPC and Co-Array Fortran have seen some success in limited application domains, because they give programmers control over data layout and local, but they are missing the kind of broad adoption that MPI enjoys.

Somewhere between Petascale and Exascale computing, due in large part to the increased exposure of on-chip parallelism, the MPI model may find its limit, as a separate process with its own address space may be too heavyweight for individual cores. Each process requires its own process state and message buffers, in addition to replicas of user level data structures that are needed by multiple processes. As the number of cores per chip increases, either hybrid parallelism or some new model is likely to take hold.

4.3.1 Automatic Parallelization

Automatic parallelization has long been the holy grail of parallel computing, allowing programmers to take advantage of parallel hardware without modifying their programs. There are many examples of successful automatic parallelization in the research literature, especially for programs dominated by arrays and loops and for machines with a modest number of processors and hardware-supported shared memory. Parallelizing compilers have been challenged by the use of languages like C that rely heavily on pointers, since they make the necessary dependence analysis difficult to impossible. The shift towards distributed memory machines in parallel computing further complicated the automatic parallelization problem, since on top of parallelization, compilers needed to automatically partition data structures across the compute node memory. Again, while progress was made on this problem for some applications, it soon became clear that programmers needed to provide the partitioning information to make this practical, and the user community, impatient for an immediate solution, instead switched to message passing.

4.3.2 Data Parallel Languages

Data parallel languages in their purest form have serial semantics, which means that every execution for a given program and input (broadly construed to include all environment information, such as random number seeds and user input) will produce the same result. Operationally, one can see all behavior of the data parallel code using a deterministic serial machine execution. Data parallel languages are distinguished from conventional serial languages through their use of operations and

assignments over aggregate data types, typically arrays. The following code fragment shows some data parallel operations arrays A and B:

```
A = B;  
B(1:N-1) = 2*B(1:N-1) - shift(B(2:N),{-1}) - shift(B(0:N-2},{1}));  
A = B;
```

The first statement assigns all of the elements in B to the corresponding elements at the same indices in A. The set of assignments within the statement can be done in any order, and because any serial or parallel order is guaranteed to give the same result, one can treat the statement semantically as if it were serial, e.g., the elements are assigned left to right. The second statement is more interesting, because it has side effects on array B that also appears on the right-hand-side. Under data parallel semantics, the entire right-hand-side is evaluated to produce an array, which we can think of as being stored in a temporary array, and that array is then assigned to the left-hand side. This preserves the serial semantics, since the side effects happen only after the expression evaluation is complete. This particular expression shows a relaxation operation (3-point stencil) in which the interior of array B, indicated by B(1:N-1), is updated with after scaling all the values by 2 and subtracting the left and right neighboring values.

There is an implicit barrier between statements, meaning that one statement cannot appear to execute before the previous has completed, although an optimizing compiler may regroup and reorder instructions as long as those reorderings do not introduce a change in semantic behavior. A compiler for the above code might divide A and B into contiguous blocks, use one per processor and allocate a “ghost region” around the blocks of B to allow for space to hold copies of neighboring values. The code could be a Single Program Multiple Data style code with one thread per processor, which performs a local assignment of the local elements of A to B, followed by communication with processing that own neighboring blocks to fill in ghost values of B, followed by a local update of B and then (without communication or synchronization) another assignment of the elements of B to A.

Data parallel languages were quite popular on SIMD architectures like the Connection Machine (CM-2) and Maspar, where the fine-grained parallelism was supported in hardware. Once clusters began dominating high end computing, data parallel languages became less popular, because the compilation problem described above was challenging in the general case. The HPF language effort was designed to address this problem by adding “layout” specifications to arrays, so that the compiler could more easily determine how to break up the problem, and the computation would follow the data using an “owner computes” rule. The HPF effort had two conflicting agendas, one being support of general computational methods, including sparse, adaptive, and unstructured, while also providing performance that was competitive with message passing style programs. This created significant language and compiler challenges, and the application community and U.S. funding agencies soon became impatient for results. There are notable examples of successful HPF applications, including two on the Earth Simulator System in Japan, which benefited from both architectural support in the form of vectors processing nodes and a significant sustained compiler effort that lasted longer after funding for HPF within the U.S. had dried up.

4.3.3 Shared Memory

The **Shared Memory Model** reflects a paradigm where parallel threads of computation communicate by reading and writing to shared variables, and there is, implicitly, a uniform cost to accessing such shared variables. The data parallel model uses shared variables to communicate,

but the term “shared memory terminology” is used here to capture programming models with an explicit form of parallelism, such as user-defined threads. The uniform cost model distinguished shared memory models from the partitioned global address space models, where there is an explicit notion of near and far memory in the programming language.

4.3.3.1 OpenMP

OpenMP is an API that supports parallel programming on shared memory machines. The API is defined by a set of compiler directives, library routines, and environment variables. These are implemented as a set of extensions to Fortran, C, and C++, which are added as comments to the base language to simplify maintenance of programs that work in both a pure serial mode without OpenMP compiler support and in parallel with such support. OpenMP was developed by an consortium of major computer hardware and software vendors with the goal of addressing the technical computing community, and it currently runs on most shared memory parallel machines. There are many features of OpenMP to support a variety of parallelization patterns, but the most common is the parallel loop. The parallel loop annotations can also include mechanisms to distinguish shared and private variables, to load balance the iterations across processors, and to perform operations like reductions, which create dependencies across loop iterations and therefore require special attention to avoid data races in the generated code. OpenMP offers more general form of parallelism besides loops, in which independent tasks (threads) operate on shared variables. OpenMP programs are typically translated to an execution layer in which threads read and write variables that live in shared memory.

OpenMP is generally considered an easier programming model than message passing, both anecdotally and in user studies.[66] There is no programmer control over data layout within OpenMP, which is key to its programming simplicity, but a limitation in scalability. There have been research efforts to run OpenMP on a software shared memory execution layer on top of distributed memory hardware, and efforts to expand OpenMP to directly support clusters, but in practice OpenMP is used only on cache-coherent shared memory hardware. Many of the largest systems today are built from shared memory nodes, and several application teams have developed hybrid versions of their codes that combine MPI between nodes and OpenMP within nodes. To date, the performance results for such hybrid codes have been mixed when compared to a flat message passing model with one process per core. While the OpenMP code avoids the costs of message construction, buffering, matching and synchronization, it often performs worse than message passing due to a combination of false sharing, coherence traffic, contention, and system issues that arise from the difference in scheduling and network interface moderation for threads as compared to processes. Well optimized OpenMP can outperform MPI on shared memory, and with sophisticated compiler and runtime techniques, it has also been demonstrated to be competitive on cluster hardware.[151] But the history of shared memory programming and message passing programming cannot be ignored: despite an expectation that hybrid programming would become popular on machines built as a distributed memory network of shared memory nodes, it has not.

There is speculation that OpenMP will see gains in popularity with multi-core systems, since most multi-core architectures today support cache-coherent shared memory. The on-chip caching should exhibit lower overheads than on multi-socket SMPs today, and the programming convenience of OpenMP continues to make it attractive for programmers newly introduced to parallelism. Whether the hybrid model will become popular for Exascale programs will depend primarily on how the performance and memory scaling tradeoffs are resolved with large number of cores per chip; the momentum in this case is in favor of a flat MPI model, which avoids the need to use two different forms of parallelism in a single application.

4.3.3.2 Threads

Many modern operating environments support the POSIX threads (PThreads), or a similar threading interface that is specific to a given operating system. This library allows for a simple thread creation mechanism as well as mutex locks, conditional signal mechanisms, and shared memory maps. A major advantage of this model, and in particular the POSIX API, is that it is available on a vast number of shared memory platforms and (unlike OpenMP) does not require compiler support. A thread library includes mechanisms to create and destroy threads, as well as locks and other synchronization mechanisms to protect accesses to shared variables. Variables allocated in the program heap are typically available to other threads, and a common programming idiom is to package the shared state in a single structure and pass a pointer to that state to each thread.

There are various special cases of threading that arise in specific languages. In several cases the language-based implementations use a lexically scoped threads mechanism, such as `cobegin/coend`, in which the lifetime of a thread is visible in the program text. The POSIX model, in contrast, gives a thread handle at the creation point, and that thread may be terminated any point in the program where the handle is available. The paired mechanisms are simpler to reason about for formally and informally, but are less expressive and require some language and compiler support to enforce the pairing. For example, the Cilk language extends C with threads and requires a compiler to translate the language syntax into lower level system threads, such as PThreads. As another example of language support for threading, the Java mechanism uses its object-orientation to support threads, whereby a class is created to contain application-specific thread state that is derived from a standard thread class, and methods on that class allow the threads to be created and managed.

An important distinction in thread systems is whether the threads are fairly scheduled in the face of long-running computations within a single thread. Fair scheduling ensures that all threads make progress independent of the number of hardware cores or hardware-supported threads that are available, and in general fairness requires some kind of preemption to ensure that one or more threads cannot dominate all available resources and leave other threads stalled. Fair scheduling is important if programs use spin locks or more subtle forms of shared memory synchronization and the number of threads exceeds hardware resources. For example, if there are two types of worker threads in a system, one producing work and the other sharing it, and those threads synchronized by accessing shared queues, then consumer threads may starve producers if they continually poll for work by reading the shared queues but are never descheduled.

Fairness and preemption comes with a cost, though, since each thread can have a significant amount of state stored in registers, which must be explicitly saved at a preemption point, and caches, which may be slowly saved as the preempting thread refills the cache. An alternate model is cooperating threading, in which the programmer is informed that threads may run to completion and need only give up control over processor resources at explicit synchronization or yield points. The Cilk language takes advantage of this by allowing programmers to specify a larger number of threads that are executed simply as either function calls or as separate threads depending on resource availability. In this way, a Cilk thread is a logical thread that only translates to a more expensive physical thread when hardware is available to hold the thread state. In this case the previous producer-consumer programmer or programs with spin locks are not supposed to work. Cooperating threading models have advantages in performance by avoiding preemption at points where locality is critical, e.g., in the middle of a dense matrix operation; they also have memory footprint advantages, since thread state may be created lazily when processors become available to run them.

4.3.4 Message Passing

The **Message Passing Interface** (MPI) is a specification of a set of library routines for message-passing. These routines handle communication and synchronization for parallel programming using the message-passing model. MPI is targeted for distributed memory machines but can also be used effectively on shared memory systems. The MPI interface was derived from a number of independent message passing interfaces that had been developed by hardware vendors and application groups to address the growing interest on distributed memory hardware in the early 90s. In this sense, MPI was the standardization of a popular programming model, rather than the introduction of a new model. Its success can be attributed to this preexisting popularity for the model, to its high performance and scalability across shared memory and distributed memory platforms, and its wide availability, which includes highly portable open source implementations like MPICH and OpenMPI.

MPI has mechanisms to send and receive contiguous blocks of memory, and while there are higher level mechanism to allow the implementation to pack and unpack non-contiguous data structures, this is typically done by programmers of higher level libraries and applications for performance considerations. Synchronous send and receive operations can easily result in deadlock, if two processes try to simultaneously send to each other. MPI therefore supports asynchronous messages, which allow send operations to complete even if the remote process is unavailable to receive, and asynchronous receive operations which can avoid buffer copying by having the user level target data structure allocated before the message arrives.

An important feature of MPI besides its point-to-point messages are collective communication operations which allow a collection of processes to perform a single operation such as a broadcast, reduction, or all-to-all communication operation. These operations can be built on top of send and received, but can also use optimized execution models or architectures, such as a special network. Collectives in MPI need not be globally performed across all processors, but sub-domains can be identified using the notion of a communicator which is defined over a subset of physical processes. This can be used to subdivide a single computation, e.g., perform reductions across rows of a matrix, and to compose together programs that were written independently. The latter is especially important with the growing interest on multi-physics simulations, in which separate models, such as an ocean and wind dynamics model in a climate simulation, are combined to model complex physical phenomenon.

MPI-2 is a second version of the MPI standard, which adds support for support for one-sided communication, dynamic processes creation, intercommunicator collective operations, and expanded IO functionality (the MPI-IO portion of the interface). While many of the MPI-2 is supported in most implementations, in particular MPI-IO, support for, and optimization of, one-sided communication has been slower. As noted above, the ubiquity of MPI relies on open source implementations, and interconnect vendors may start with these open source version and optimize certain features of the implementation.

MPICH is an implementation of the MPI-1 specification, while **MPICH2** supports the expanded MPI-2 interface. MPICH is one of the oldest and most widely used MPI implementations, and benefits from continued, active development as well as wide scale usage in research and production environments. **OpenMPI** is a relatively new implementation of the MPI-2 specification. It is an open source project maintained by academic, research, and industry partners. The core development team was composed of members of many different MPI implementations, and it represents a consolidation of their experience and expertise. The focus for OpenMPI has been on increased reliability. Specific features include support for dynamic process spawning, network and process fault tolerance, and thread safety.

4.3.5 PGAS Languages

A **Partitioned Global Address Space** (PGAS) combines some of the features of message passing and shared memory threads. Like a shared memory model, there are shared variables including arrays and pointer-based structures that live in a common address space, and are accessible to all processes. But like message passing, there the address space is logically “partitioned” so that a particular section of memory is viewed as “closer” to one or more processes. In this way the PGAS languages provide the necessary locality information to map data structure efficiently and scalably onto both shared and distributed memory hardware. The partitioning provides different execution and performance-related characteristics, namely fast access through conventional pointers or array indexes to nearby memory and slower access through global pointers and arrays to data that is far away. Since an individual process may directly read and write memory that is near another process, the global address space model directly supports one-sided communication: no participation from a remote process is required for communication. Because PGAS languages have characteristics of both shared memory threads and (separate memory) processes, some PGAS languages use the term “thread” while others use “process.” The model is distinguishable from shared memory threads such as POSIX or OpenMP, because the logical partitioning of memory gives programmers control over data layout. Arrays may be distributed at creation time to match the access patterns that will arise later and more complex pointer-based structures may be constructed by allocating parts in each of the memory partitions and linking them together with pointers.

The PGAS model is realized in three decade-old languages, each presented as an extension to a familiar base language: **Unified Parallel C** (UPC)[31] for C; **Co-Array Fortran** (CAF)[112] for Fortran, and Titanium[160] for Java. The three PGAS languages make references to shared memory explicit in the type system, which means that a pointer or reference to shared memory has a type that is distinct from references to local memory. These mechanisms differ across the languages in subtle ways, but in all three cases the ability to statically separate local and global references has proven important in performance tuning. On machines lacking hardware support for global memory, a global pointer encodes a node identifier along with a memory address, and when the pointer is dereferenced, the runtime must deconstruct this pointer representation and test whether the node is the local one. This overhead is significant for local references, and is avoided in all three languages by having expression that are statically known to be local, which allows the compiler to generate code that uses a simpler (address-only) representation and avoids the test on dereference.

These three PGAS languages used a static number of processes fixed at job start time, with identifiers for each process. This Single Program Multiple Data (SPMD) model results in a one-to-one mapping between processes and memory partitions and allows for very simple runtime support, since the runtime has only a fixed number of processes to manage and these typically correspond to the underlying hardware processors. The languages run on shared memory hardware, distributed memory clusters, and hybrid architectures. On shared memory systems and nodes within a hybrid system, they typically use a thread model such as Pthreads for the underlying execution model.

The distributed array support in all three languages is fairly rigid, a reaction to the implementation challenges that plagued the High Performance Fortran (HPF) effort. In UPC distributed arrays may be blocked, but there is only a single blocking factor that must be a compile-time constant; in CAF the blocking factors appear in separate “co-dimensions;” Titanium does not have built-in support for distributed arrays, but they are programmed in libraries and applications using global pointers and a built-in all-to-all operation for exchanging pointers. There is an ongoing tension in this area of language design between the generality of distributed array support and the desire to avoid significant runtime overhead.

Each of the languages is also influenced by the philosophy of their base serial language. Co-Array Fortran support is focused on distributed arrays, while UPC and Titanium have extensive support for pointer-based structures, although Titanium also breaks from Java by adding extensive support for multidimensional arrays. UPC allows programmers to deconstruct global pointers and to perform pointer arithmetic such as incrementing pointers and dereferencing the results. Titanium programs retain the strong typing features of Java and adds language and compiler analysis to prove deadlock freedom on global synchronization mechanisms.

4.3.6 The HPCS Languages

As part of the Phase II of the DARPA HPCS Project, three vendors—Cray, IBM, and SUN—were commissioned to develop new languages that would optimize software development time as well as performance on each vendor’s HPCS hardware being developed over the same time period. Each of the languages — Cray’s Chapel⁶,/indexChapel IBM’s X10⁷, and Sun’s Fortress⁸—provides a global view of data (similar to the PGAS languages), together with a more dynamic model of processes and sophisticated synchronization mechanisms.

The original intent of these languages was to exploit the advanced hardware architectures being developed by the corresponding vendors, and in turn to be particularly well supported by these architectures. However, in order for these languages to be adopted by a broad sector of the community, they will also have to perform reasonably well on other parallel architectures, including the commodity clusters on which much parallel software development takes place. (And, in turn, the advanced architectures will have to run “legacy” MPI programs well in order to facilitate the migration of existing applications.)

The goal of these language efforts was to improve the programmability of HPC systems. This included both lowering the barrier to entry for new parallel programmers and making experienced programmers more productive. The HPCS languages all provide high level language support for abstraction and modularity using object-orientation and other modern language features, augmenting these with novel ideas for creating massive amounts of parallelism. The HPCS languages share some characteristics with each other and with the PGAS languages: the use of global name space, explicit representation of localities, and syntactic distinction of local and global data access. They all differ from the previously-described PGAS languages because they allow for dynamic parallelism and (in some cases) data parallelism, rather than a static number of threads. These languages require sophisticated compiler and runtime techniques, and in each case the vendors have developed at least prototype implementations that demonstrate the feasibility of implementation, although not necessarily at scale. Work on the implementations and analysis of productivity and performance benefits of the languages is ongoing.

Chapel is a parallel programming language developed by Cray. It supports data parallel programming and builds on some of the successful results from the ZPL languages. Chapel is not an extension of an existing serial languages, but addresses some of the limitations of the serial languages in addition providing parallelism support. A goal of Chapel is to provide better abstractions for separating algorithmic logic and data structure implementation and to provide programmers with a global view of the computation, rather than programming on a per-thread basis. Chapel uses data parallelism: data is distributed over memory partitions known as locales, and execution location can be controlled. Chapel is designed with features of the Cray’s Cascade system in mind, including hardware support for a global address space, but is currently implemented

⁶<http://chapel.cs.washington.edu/>

⁷http://domino.research.ibm.com/comm/research_projects.nsf/pages/x10.index.html

⁸<http://projectfortress.sun.com/Projects/Community/>

on top of the GASNet communication layer and using a source-to-source translation system which make it portable across many existing machines.

Fortress is a parallel programming language designed at Sun. Like Chapel, Fortress uses a new syntax and semantics, rather than building on an existing serial language. Fortress uses a shared global address space but generalizes it through a hierarchical notion of partitioning, which could prove useful on machines with hierarchical parallelism and for composing separately-written modules. In an effort to expose maximum parallelism in applications, Fortress loops and argument evaluations in function calls are both parallel by default. The focus is on extensibility, and there is a novel type system for building libraries and allowing them to be analyzed and optimized by the compiler. The language takes advantage of the support for modularity and extensibility by implementing a small core language directly and then supporting a large set of standard libraries, in the same spirit as the Java libraries. Including the libraries, Fortress also provides support for matrices and vectors, with static checking for properties, such as physical units or boundary conditions, included.

X10 is an extension of the Java programming language. Java was chosen as a base language for its type safety features, object-oriented style, and familiarity among developers. X10 supports distributed object-oriented programming. The target architecture for X10 are low and high end systems comprised of multi-core SMP nodes. With X10, the memory is logically partitioned into locales and data is explicitly distributed by programmers to those locales. Computations can be explicitly assigned to particular locales which is implemented as remote function invocation. The first implementation was done for shared memory with shared task queues, but a implementation on top of the LAPI communication layer is also under development.

4.4 Today's Microprocessors

In this section we review briefly the state of current microprocessor chips which form the basis of all classes of computing from embedded to supercomputers. In most cases, historical data is combined with data from the ITRS Roadmap[13] to show trends, and more importantly, where trends break. The Roadmap data goes back to 2004 to overlap with the historical data and give an indication of how accurate the ITRS projections have been in the past.

4.4.1 Basic Technology Parameters

Perhaps the single parameter that most drives the semiconductor industry in general and the microprocessor vendors in particular, is the continued decline in the **feature size** of the transistors that make up the logic and storage circuits of a microprocessor. Figure 4.2 diagrams this parameter over time, with historical data from real microprocessors (labeled at the time of their first release) combined with projections from the ITRS roadmap for transistors to be used in logic circuits. As the trend line shows, the leading edge devices have been improving by a factor of about 0.88 per year. Today, leading edge production is being done at the 65 nm node, with experimental chips being produced at around 40 nm feature size.

A related characteristic is Figure 4.3, the density of transistors on a CMOS die over time, measured in millions of transistors per sq. mm. There are two distinct trend lines in this figure: that for the ITRS projections at a CAGR of 1.28 per year, and the historical trend of a higher 1.5 per year. A possible reason for this discrepancy is the huge growth in transistor-dense cache structures during the time of the single core microprocessors. This is buttressed by Figure 4.4 which shows a historical CAGR of 1.82 - significantly in excess of the growth in transistor density. How this will continue into the era of multi-core designs is unclear.

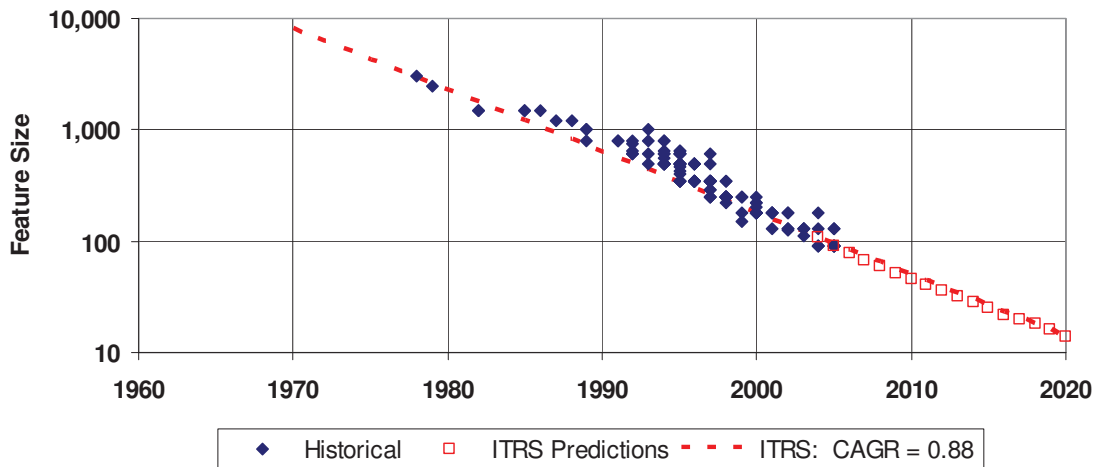


Figure 4.2: Microprocessor feature size.

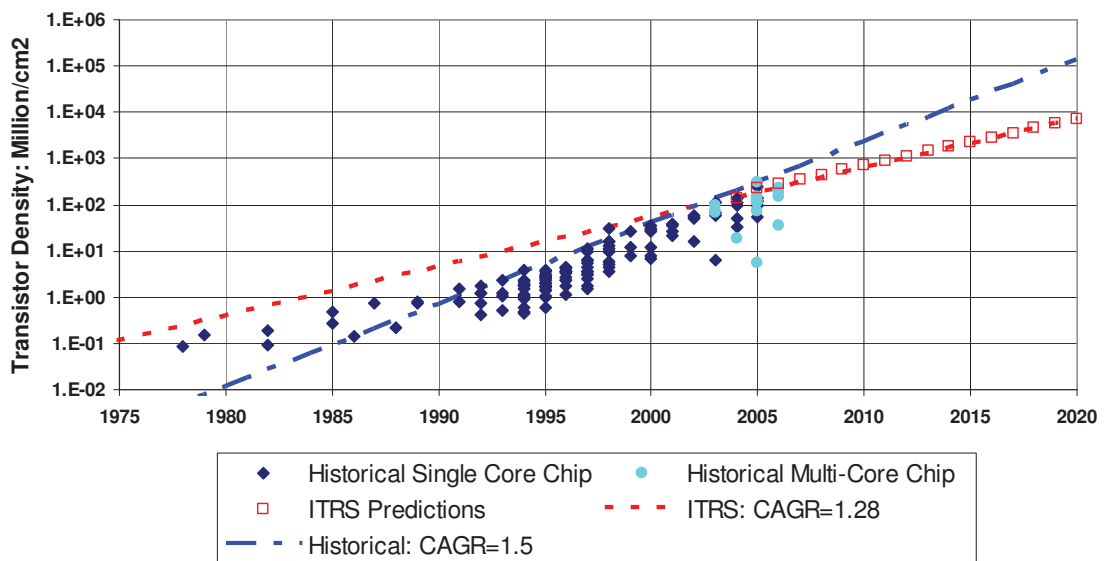


Figure 4.3: Microprocessor transistor density.

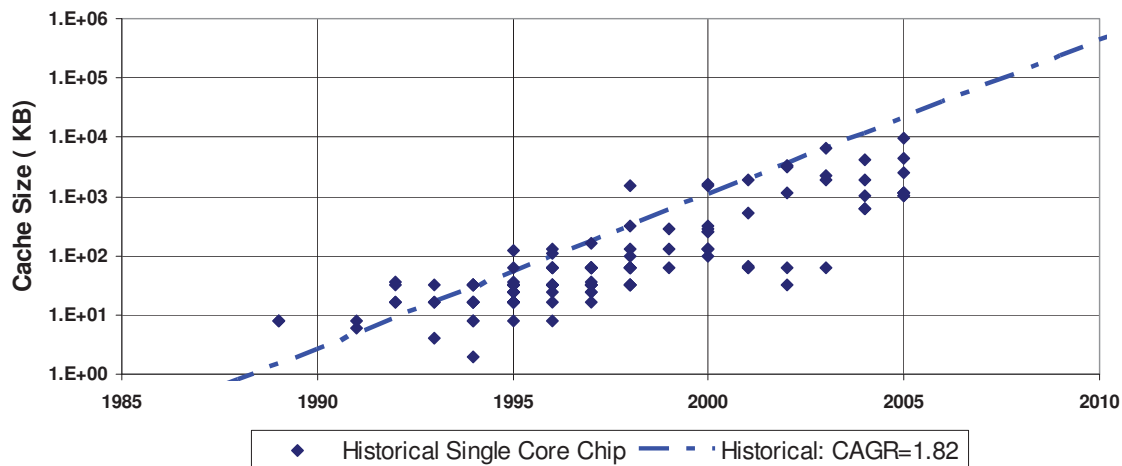


Figure 4.4: Microprocessor cache capacity.

4.4.2 Overall Chip Parameters

Two top level related chip-level parameters are the actual die size used for the chip (Figure 4.5) and the actual number of transistors on that die (Figure 4.6). The former exhibited a continual rise in size until 1995, after which the maximum die size varies, and is projected to continue to vary, in the 200-33 sq. mm range. The latter is pretty much a direct product of the die size of Figure 4.5 and the density of Figure 4.3, and an interpretation is similar to that for those figures.

Of more direct importance to this study is the voltage level used for the microprocessors (Figure 4.7). The V_{dd} curve shows a **constant voltage** at 5 volts until about 1993, when **constant field scaling** was employed and V_{dd} could be decreased very rapidly. As will be discussed later, this decrease balanced several other factors that affected the power of the die, and allowed faster chips to be deployed. After about 2000, however, this rapid decrease slowed, mainly because of minimums in the threshold voltages of CMOS transistors. Looking into the future, V_{dd} is projected to flatten even more, with a relatively tight range between what is used for high performance parts and that used for low power embedded parts.

Likewise, the decrease in transistor feature size led directly to faster transistors, which in turn led to increasing clock rates, as pictured in Figure 4.8. Up through the early 2000s' the historical clock rate increased at a CAGR of 1.3X per year. After 2004, the actual parts produced stagnated at around 3 GHz, below even a decreased ITRS projection.

It is important to note that these ITRS clock projections were computed as a maximum number assuming that the logic stays fixed at a 12 gate per pipeline stage delay, and the clock is raised up to a rate commensurate with the intrinsic improvement in the individual transistor delay. As will be discussed later, for power reasons, the actually implemented clock rates are now considerably less than this curve.

Finally, Figures 4.9 and 4.10 give both the power dissipated per die and the power density - the power dissipated per unit area. Both numbers went through a rapid increase in the 1990s, and then hit a limit. The maximum power that could be cooled at reasonable expense for a die destined for a high volume market was in the order of 100+ watts. Once this was reached, something had to be done to reduce the power, and that was done by reducing the clock - regardless of how fast the individual devices could go.

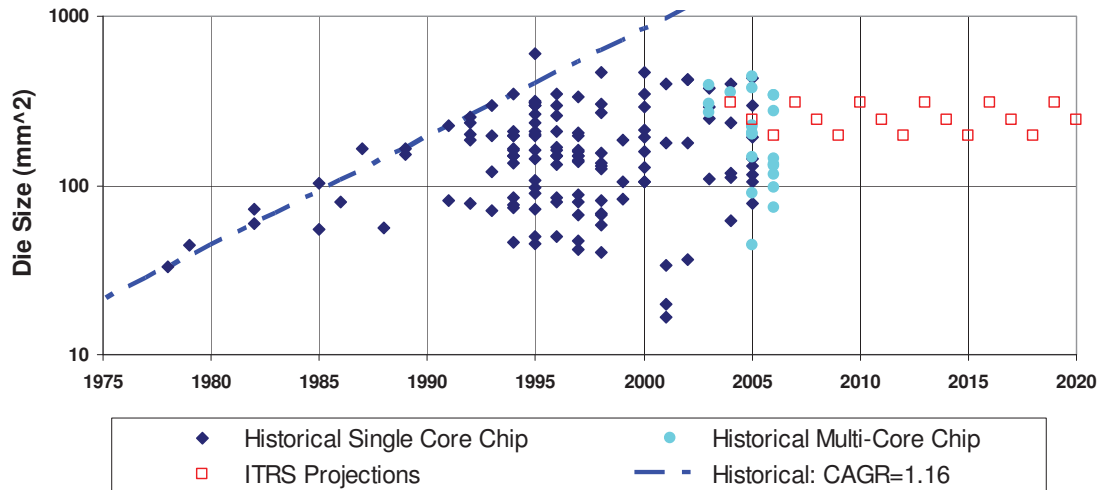


Figure 4.5: Microprocessor die size.

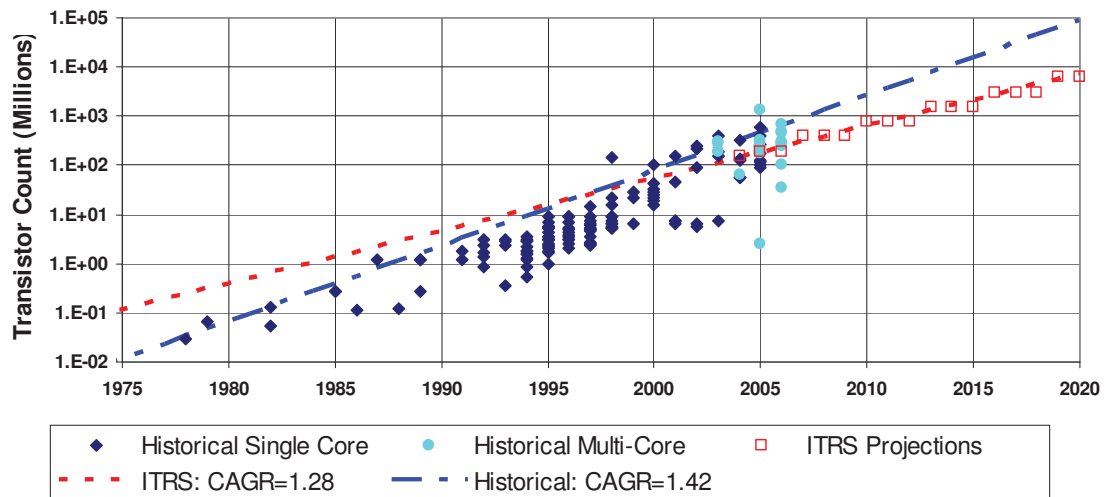


Figure 4.6: Microprocessor transistor count.

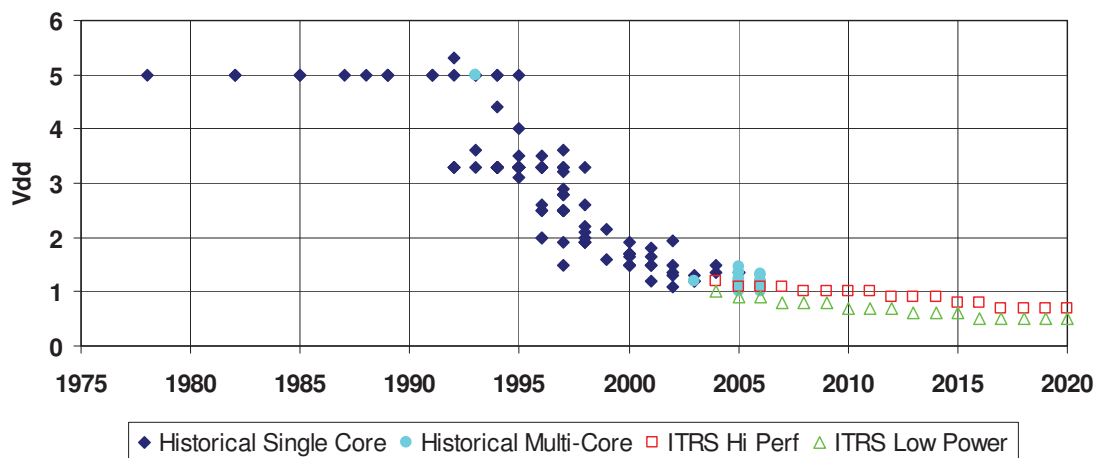


Figure 4.7: Microprocessor V_{dd} .

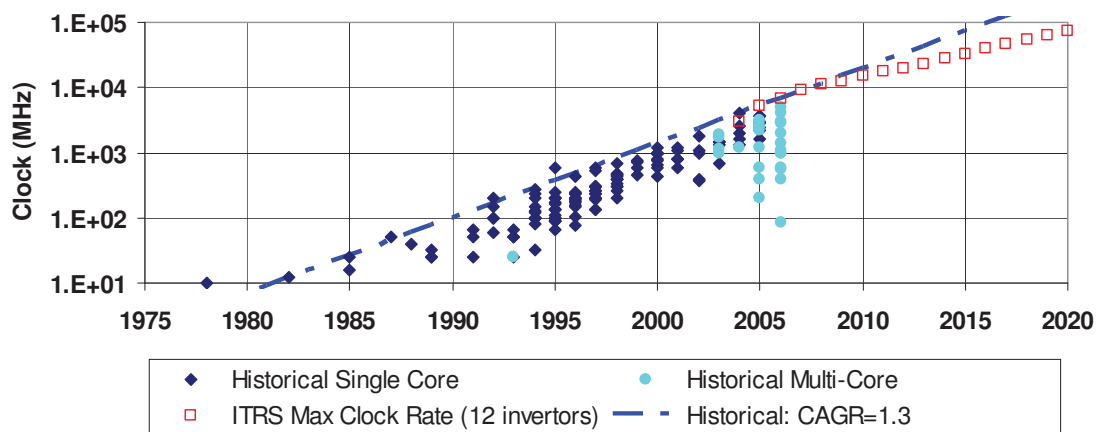


Figure 4.8: Microprocessor clock.

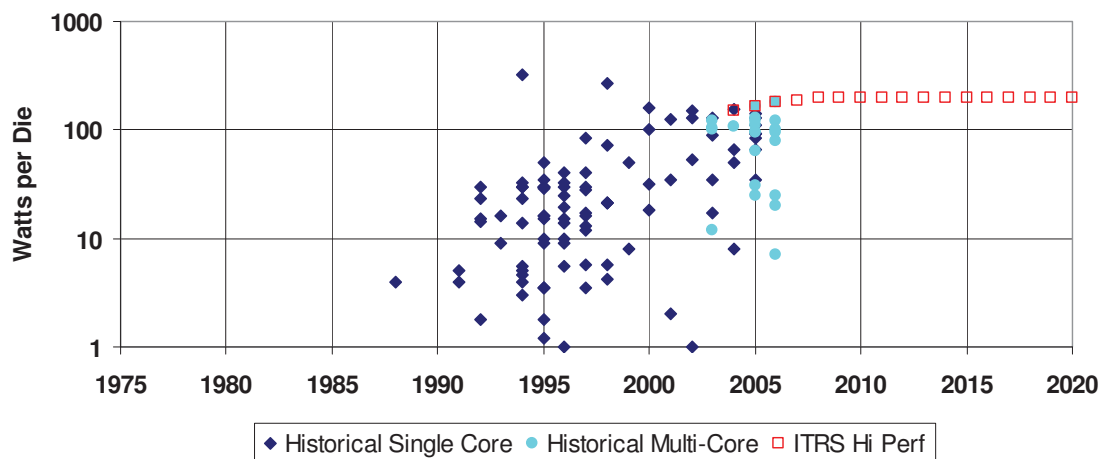


Figure 4.9: Microprocessor chip power.

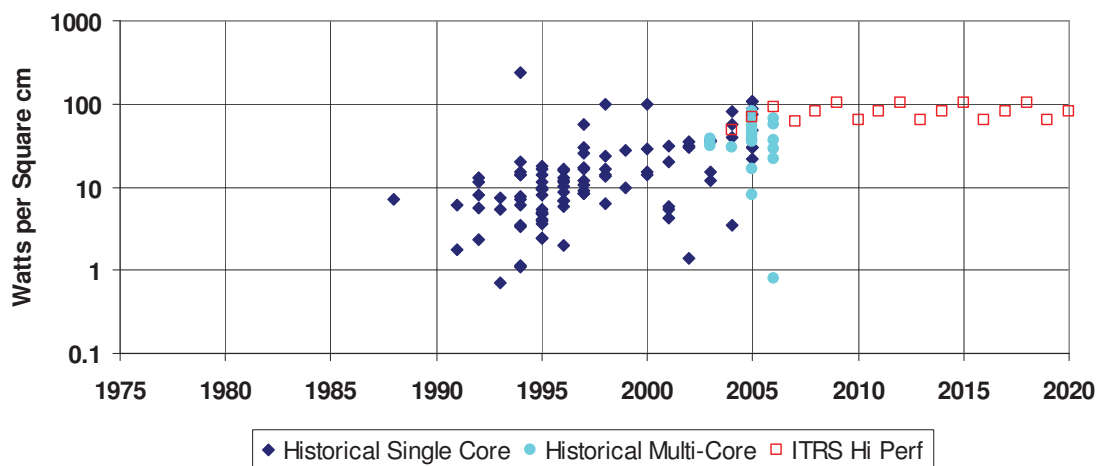


Figure 4.10: Microprocessor chip power density.

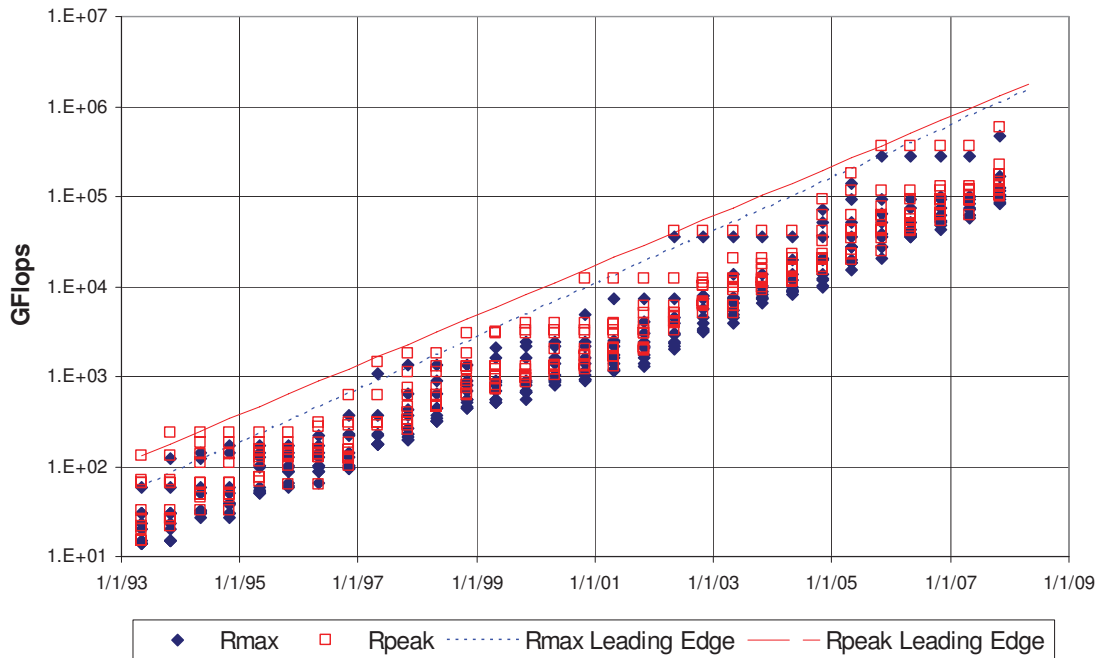


Figure 4.11: Performance metrics for the Top 10 supercomputers over time.

4.4.3 Summary of the State of the Art

Microprocessor design has ridden Moore's Law successfully for decades. However, the emergence of the power wall has fundamentally changed not only the microarchitecture of microprocessors (with the rise of multi-core as discussed in Section 4.1.1), but also their actual computational rates (as evidenced by the flattening of clock rates). These trends will continue.

4.5 Today's Top 500 Supercomputers

The **Top 500**⁹ is a list of the top 500 supercomputers ("data-center sized") in the world as measured by their performance against the **Linpack** dense linear algebra benchmark, with a metric of floating point operations per second (flops). It has been updated every 6 months (June and November) since 1993, and as such can give some insight into the characteristics and trends of one class of both data center scale hardware systems and high end floating point intensive applications.

4.5.1 Aggregate Performance

Figure 4.11 gives two performance metrics for the top 10 from each list since 1993: R_{peak} is the theoretical peak performance of the machine in gigaflops, and R_{max} is the best measure floating point count when running the Linpack benchmark. The top 10 were chosen for study, rather than all 500, because they tend to represent the best of the different architectural tracks without introducing a lot of duplication based on replication size alone. Even so, as can be seen, the spread in performance is uniformly about an order of magnitude.

⁹<http://www.top500.org/>

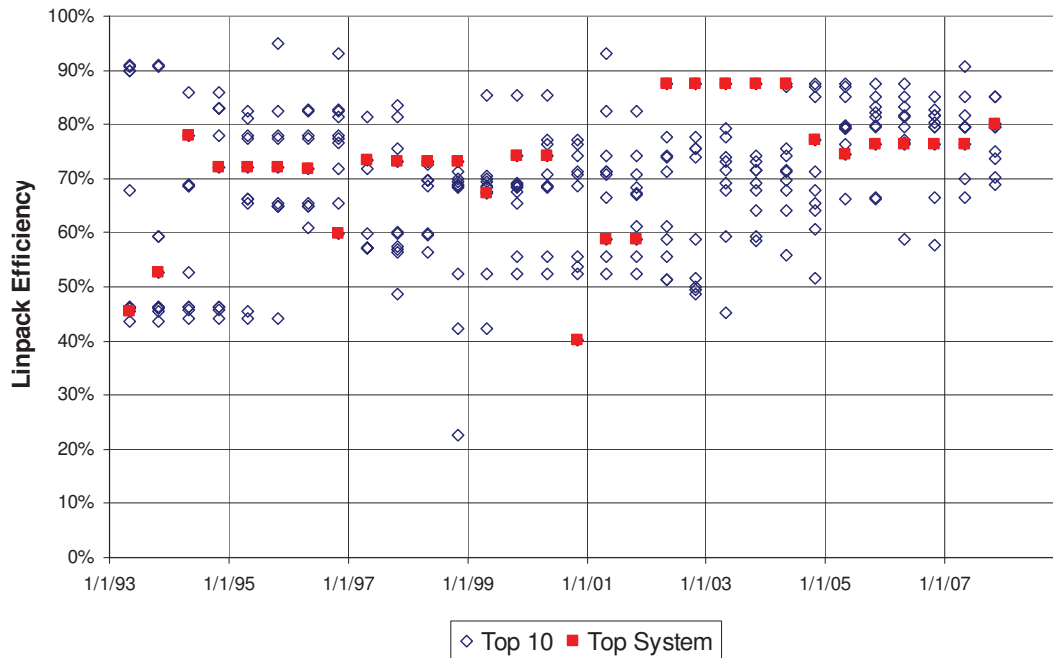


Figure 4.12: Efficiency for the Top 10 supercomputers while running Linpack.

As can be seen, the CAGR for R_{peak} is about 1.89 and for R_{max} it is 1.97. This translates into about a 10X growth every 41-43 months, or 1000X in between 10 and 11 years. If 2010 is nominally the year of the first sustained petaflops R_{peak} , then if these rates were sustainable, it will be 2020 for the first exaflops machine.

4.5.2 Efficiency

We define **efficiency** as the ratio of the number of useful operations obtained from a computing system per second to the peak number of operations per second that is possible. Figure 4.12 graphs this metric as the ratio of R_{max} to R_{peak} , with the top system in each list indicated as a square marker. The major observations from this chart are that:

- the range has historically been between 40 and 90
- there has been a tightening of the lower end over the last 5 years,
- the efficiency of the top performing system has not always been the highest, but has been in a tighter range from 70 to 90

4.5.3 Performance Components

The performance of these machines can be expressed as the product of three terms:

- **Parallelism:** the number of separate “processor” nodes in the system, each nominally capable of executing a separate thread of execution (none of the machines on the Top 500 are to date multi-threaded).

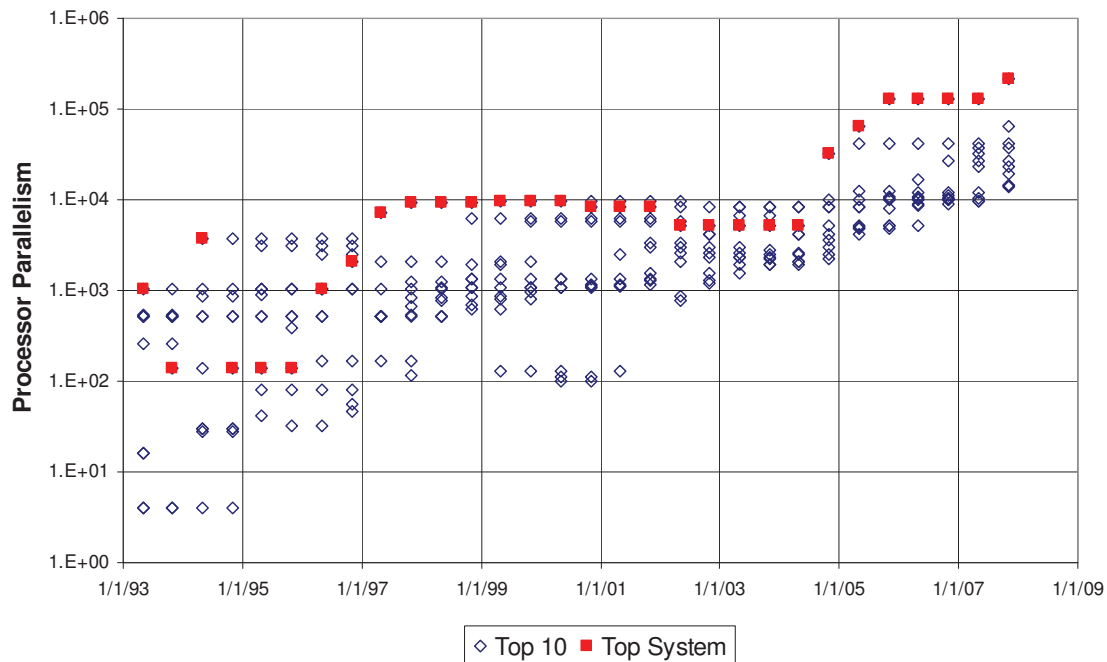


Figure 4.13: Processor parallelism in the Top 10 supercomputers.

- **Clock:** the rate at which operations can be performed in any single processor.
- **Thread Level Concurrency (TLC):** this is equivalent to the number of separate operations of interest (floating point for these Linpack-based kernels) that can be executed per cycle.

We note that if we look at the activities in each machine cycle, then the total number of processors times the maximum TLC gives some sense to the “overall concurrency” in a system, that is the total number of separate hardware units capable of parallel operation.

Each of these topics is discussed separately below. However, the clear take away from them is that since 1993, the performance gains have been driven primarily by brute force parallelism.

4.5.3.1 Processor Parallelism

Parallelism is the number of distinct threads that makes up the execution of a program. None of the top systems to date have been multi-threaded, so each processor as reported in the Top 500 list corresponds to a single thread of execution, or in modern terms a “single core.” Figure 4.13 then graphs this number in each of the top 10 systems over the entire time frame, with the top system highlighted.

The key observation is that the top system tended to lead the way in terms of processor parallelism, with the period of 1993 to 1996 dominated by systems in the 100 to 1000 region, 1996 through 2003 in the 10,000 range, and 2004 to now in the 100,000 range.

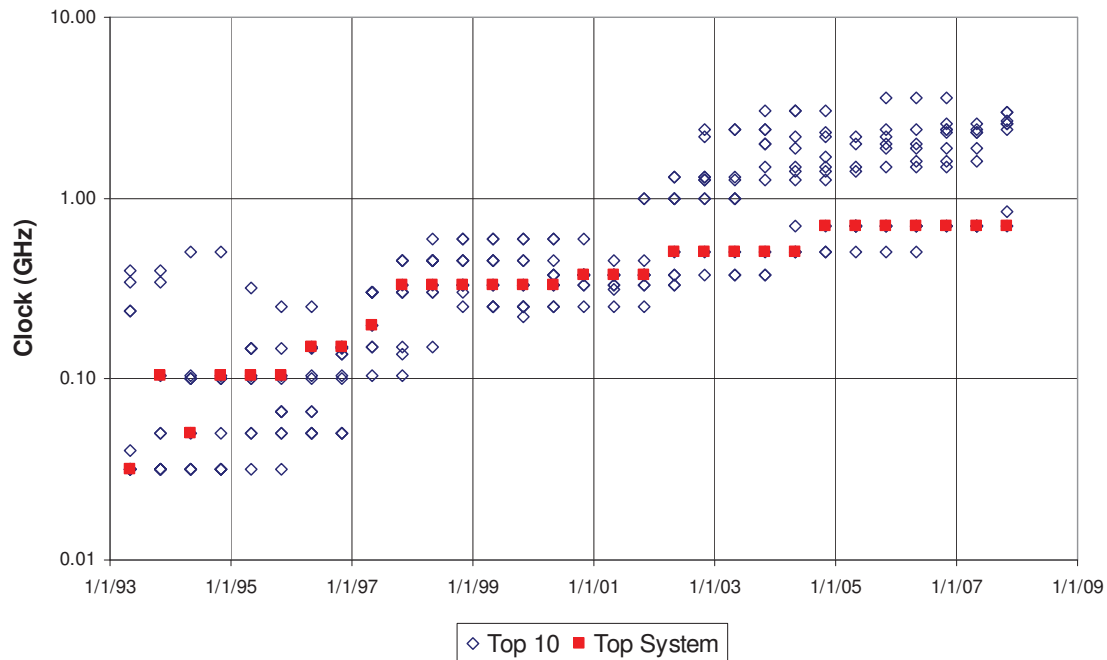


Figure 4.14: Clock rate in the Top 10 supercomputers.

4.5.3.2 Clock

The **clock** is the rate at which operations can be performed in any single processor. Figure 4.14 graphs this for the top 10 systems, with the top system highlighted as before. Here the growth rates are much different that for parallelism. While the highest clock rates for Top 10 systems have in general been in line with the best of then-leading edge technology, the clock rate growth for the top system has been extremely modest, with a range of from 500 to 700 MHz for well over a decade.

4.5.3.3 Thread Level Concurrency

Thread Level Concurrency (TLC) is an attempt to measure the number of separate operations of interest that can be executed per cycle. For the Top 500 to date the operation of interest has been floating point operations (based on the Linpack-based kernels). It is computed as the performance metric (R_{max} or R_{peak}) divided by the product of the number of processors and the clock rate as given in the lists.

TLC is meant to be similar to the **Instruction Level Parallelism (ILP)** term used in computer architecture to measure the number of instructions from a single thread that can either be issued per cycle within a microprocessor core (akin to a “peak” measurement), or the number of instructions that are actually completed and retired per second (akin to the sustained or “max” numbers of the current discussion). Figure 4.15 graphs both the peak and the max for the top 10 systems, with the top system highlighted as before.

As can be seen, these numbers reflect the microarchitecture of underlying processors. Those systems with peak numbers on the 16 to 32 range have for the most part been vector machines. Virtually all of the rest have been 4 or less, both in max and peak, and correspond to more or less

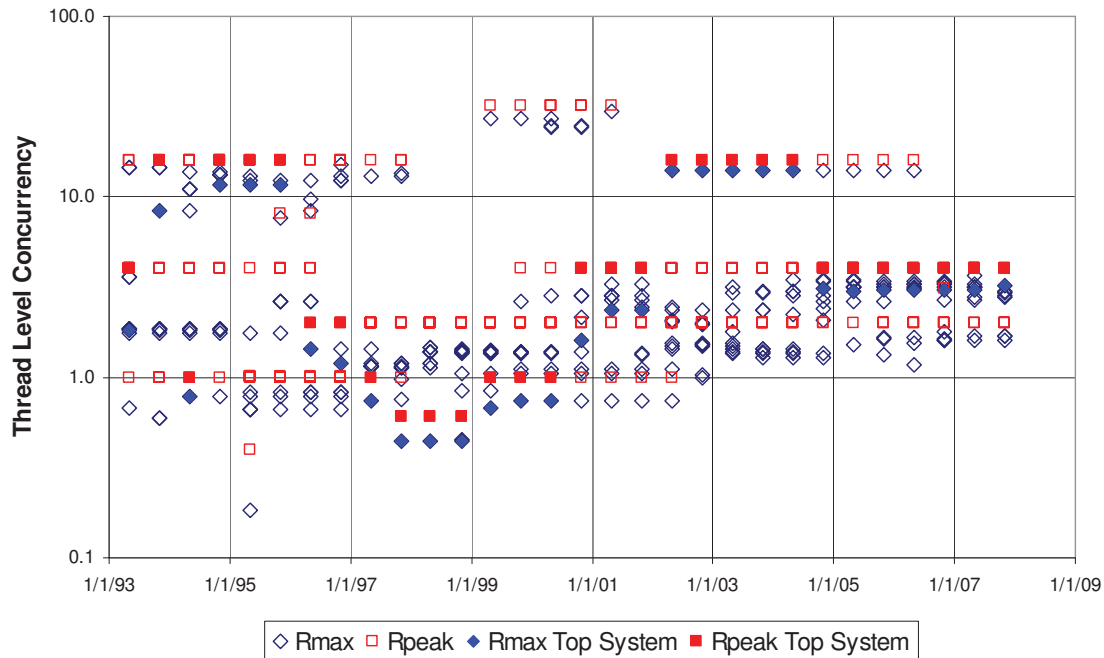


Figure 4.15: Thread level concurrency in the Top 10 supercomputers.

conventional microprocessor designs with just a very few floating point units, each often counted as being capable of executing up to two operations per cycle (a **fused multiply-add**).

With the exception of the Earth Simulator, virtually all of the top systems over the last decade have been these relatively low TLC designs.

4.5.3.4 Total Concurrency

Total concurrency is the total number of separate operations of interest that can be computed in a system at each clock cycle. For the Top 500 these operations are floating point, and such a measure thus reflects (within a factor of 2 to account for fused multiply-add) the total number of distinct hardware units capable of computing those operations.

For our Top 10 list, this metric can be computed as the number of processors times the peak TLC. Figure 4.16 graphs these numbers, with the top system highlighted as before. Unlike just the processor parallelism of Section 4.5.3.1 (which is very stair-stepped), and the clock of Section 4.5.3.2 and the TLC of Section 4.5.3.3 (which have at best very irregular trends), the Top 1 system tends to ride at the very top of the curve, and to advance in a monotonic fashion. To see this more clearly, a trend line is included that touches the transitions of the top system almost perfectly. The CAGR for this line is about 1.65, meaning that a 10X increase in concurrency is achieved every 4.5 years, and a 1000X in 13.7 years. We note that this CAGR is equivalent to about 2.17X every 18 months, which is somewhat above Moore's Law, meaning that the rate in increase in separate computational units is increasing faster than the number that can be placed on a die, implying that relatively more chips are being added to the peak systems of each succeeding generation.

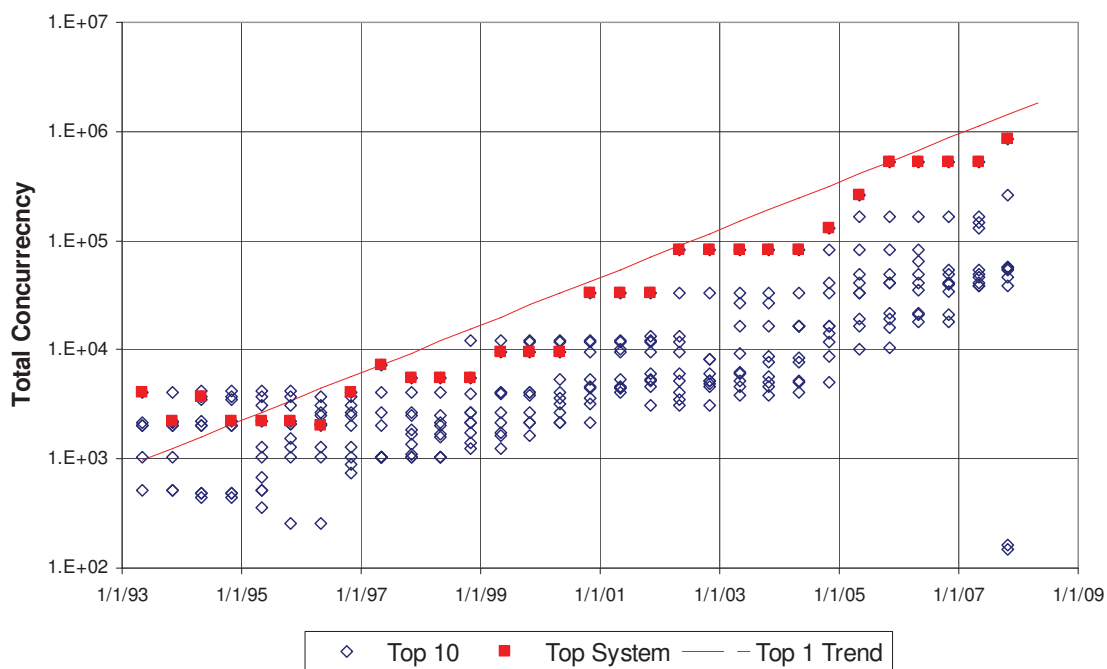


Figure 4.16: Total hardware concurrency in the Top 10 supercomputers.

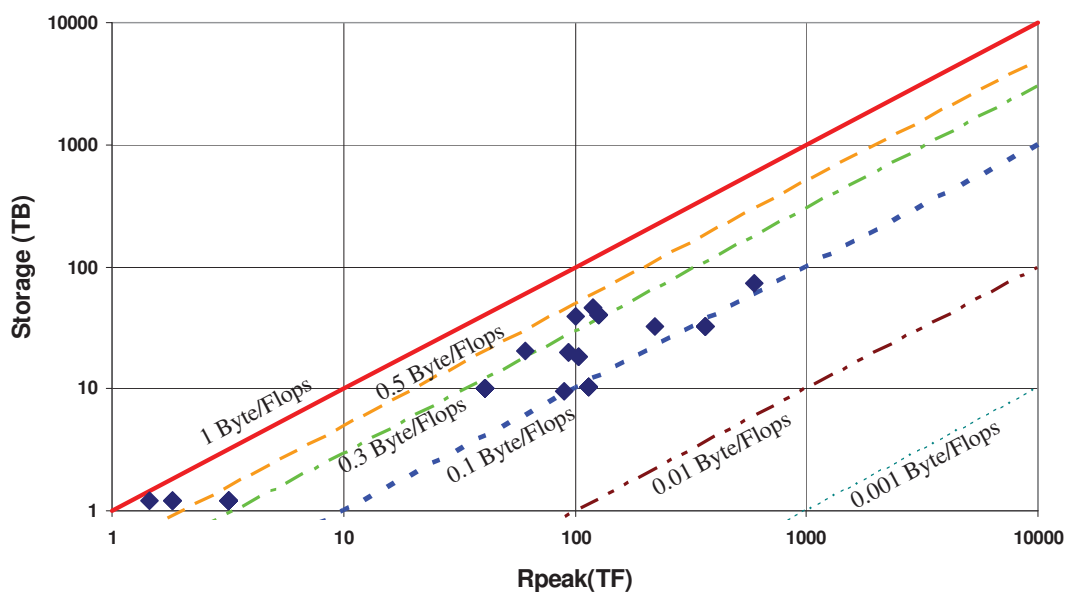


Figure 4.17: Memory capacity in the Top 10 supercomputers.

4.5.4 Main Memory Capacity

Another important aspect of such machines that has a direct affect on the actual problems that can be placed on them is the amount of directly addressable memory that is available to hold the data. Figure 4.17 plots main memory capacity versus R_{peak} for a set of the most recent top 10 machines. Also included are lines of “bytes per flops.”

As can be seen, virtually all the recent machines are clustered in the 0.1 to 0.5 bytes per flops range. If these trends continue, we would thus expect that the near term Petascale systems may have main memory capacities in the 100-500TB range.

When we look at future machines, two points may be worth considering. First, most of the current Top 10 machine architectures are clusters that employ message passing. Thus there may be some extra memory consumed in duplicate programs, and in data space for data exchange (i.e. “ghost node” data) that might not be present in a shared address space machine. Second, the x-axis in Figure 4.17 is R_{peak} , not R_{max} . The latter reflects at least an executing program, and not just a simple “100% busy FPU” calculation. With real efficiencies in the 70-90%, the storage per “useful” flop is actually somewhat higher than shown.

This page intentionally left blank.

Chapter 5

Exascale Application Characteristics

This chapter attempts to develop a characterization of a class of applications that are liable to be significant to Exascale systems. Because of our long history of difficulties inherent in porting such applications to the largest computing systems, and because the middle departmental class of Exascale systems is still at the scale as today's largest, the bulk of the discussion will be on applications relevant to data center class systems.

At these scales, the overall performance of a system is a complex function of many parameters of the system's design, such as logic clock speeds, latencies to various memory structures, and bandwidths. Our discussion will attempt to analyze effects on performance due to changes in one or multiple parameters.

Section 5.1 first defines a graphical representation that will be used for exploring these effects. Section 5.2 describes the concepts of balance and the von Neumann bottleneck in terms of applications and performance. Section 5.3 then describes a typical application of significance today, and how a machine that is "balanced" in its design parameters might behave as those parameters are changed. Section 5.4 briefly discusses how different classes of applications may be composed of more basic functionality. Section 5.5 then performs an analysis of several strategic applications of today that are particularly sensitive to memory system parameters.

Section 5.6 then focuses on understanding what it means to "scale performance" of applications by 1000X. Section 5.7 then looks at several applications and how they in fact scale to higher levels of performance.

Section 5.8 then summarizes what this all means to a potential Exascale program.

5.1 Kiviat Diagrams

The performance of real applications on real computers is a complex mapping between multiple interacting design parameters. The approach used here to describe such interactions is through use of **Kiviat diagrams**, or "radar plots." In such diagrams a series of radial axes emanate from the center, and a series of labeled concentric polygonal grids intersect these axes. Each axis represents a performance attribute of the machine that might be individually improved, such as peak flops, cache bandwidth, main memory bandwidth, network latency etc. Each polygon then represents some degree of constant performance improvement (usually interpreted here as a "speedup") of the application relative to some norm, with '1' (the degenerate polygon at the center) representing a baseline performance of the machine with no modifications. The units of the axis are normalized improvement.

A dark lined polygon drawn on this diagram then represents the effects on application per-

formance resulting from improving the design parameters associated with each axis by some fixed amount, such as 2X. Thus by moving from axis to axis, and seeing where this dark line lies in relation to the labeled grid polygons, one can tell the relative effect of each component taken in isolation. Vertices of this dark polygon that are further out from the origin than other vertices thus correspond to attributes where the relative change has a larger effect on overall performance.

In many cases, some axes are labeled for combinations of parameters. In such cases, the resultant measurement is performance when *all* those parameters are simultaneously improved by the same amount.

5.2 Balance and the von Neumann Bottleneck

The term “balanced design” refers to a design where some set of resources, which individually “cost” about the same, are used at about the same levels of efficiency, so that adding more of one resource without adding the same amount of the others adds very little to overall system performance. In terms of computing, such balancing acts typically revolve around the computational versus memory access resources. The von Neumann bottleneck, i.e. limited data transfer rate between the processor and memory compared to the amount of memory, has been a performance limiter since the inception of the digital computer. From Wikipedia: the term “von Neumann bottleneck” was coined by John Backus in his 1977 ACM Turing award lecture. According to Backus:

Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it.

This bottleneck is exacerbated by modern architectural trends, and is particularly irksome in scientific computing that consists primarily in evaluating mathematical expressions exemplified by a simple example: $A = B + C$. To carry out this calculation, the computer must fetch the arguments B and C into the processor from wherever they reside in the memory, then carry out the mathematical operation, then store the result A back into memory; unfortunately the fetching and storing steps can take several of orders of magnitude longer than the mathematical operation (+) on today’s processors.

This imbalance is growing as an indirect result of Moore’s Law, which states that the density of transistors on a chip doubles every 18 months or so. The resulting smaller logic and shorter signaling distances has primarily been used to enable ever higher processor clock frequencies, with resulting faster processors for carrying out mathematical operations, while the absolute distance to memory off-chip remains about the same, and thus the *relative* time to access this data (time measured in processor clock cycles per access) becomes greater with every turn of Moore’s Law. This phenomenon has been termed “**red shift**” because, analogous to our expanding universe, computers seem to recede in relative distance to their own local memory and storage, and each other, with every turn of Moore’s Law. Another implication of red shift is that modern computers spend most of their time moving data, rather than performing mathematical operations, when running today’s memory intensive applications. We also observe that more and more applications

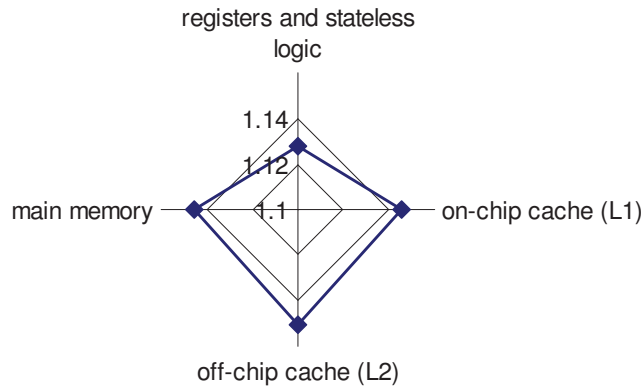


Figure 5.1: Predicted speedup of WRF “Large”.

become memory-bound in the same sense (i.e. they spend the greatest percentage of their time moving data).

As a historical context: the Cray XMP[157], the world’s fastest supercomputer in 1983-1985, was a “balanced machine” in that it could perform 2 fetches and 1 store from main memory, as well as up to 2 floating-point mathematical operation, per cycle. Thus operations such as vector inner products could run at the maximum memory rate, and yet still utilize 100% of the floating point units capabilities.

In contrast, today’s fastest supercomputer, the IBM BG/L[48], can accomplish 4 floating-point mathematical operations in 1 (much shorter) cycle yet requires around 100 such cycles to fetch just 1 argument from memory, at a rate of somewhere between 0.5 and 0.8 bytes per cycle if the data must come from local memory. This explains why the XMP would spend about 50% of its time moving data on a memory intensive code, but BG/L may spend 99% of its time moving data when executing the same code.

5.3 A Typical Application

Let us consider a “typical” scientific computation of today. As depicted in Figure 5.1, Datastar, SDSC’s IBM Power4-based supercomputer, is approximately a “balanced” machine for the **WRF (Weather Research and Forecasting)** benchmark when run on 256 CPUs with various system parameters are doubled relative to IBM Power4 on a well-known input (the HPCMO “Large” test case). This balance comes from the observation that the predicted speedup is about the same factor (to one decimal place of precision) if any one of the following occur:

1. the arithmetic logic and registers are clocked faster, by a factor of 2.
2. the latency to the on-chip L1 cache is halved without improving anything else,
3. the latency to the off-chip L2 cache is halved, again exclusive of other improvements,
4. the latency to main memory halved.

As can be seen by the dark polygon of Figure 5.1, the relative performance improvement for any one of these individual improvements is within a percent or two of only 14%. The reason for

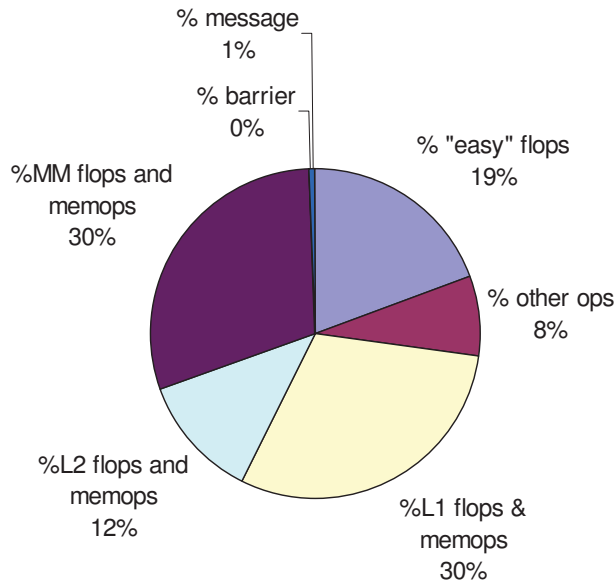


Figure 5.2: Time breakdown of WRF by operation category.

this balance is clear from consulting Figure 5.2 below which shows the time fraction spent by WRF in executing operations in different categories on DataStar, defined as follows:

- “Easy” flops that reuse data in registers (there is no associated memory fetch to get in the way of their completion).
- “Other ops” include address arithmetic, branch compares and etc.
- “L1 flops and memops” (memory operations) are either fetches from L1, or flops that cannot be issued until such a fetch completes.
- “L2 flops and memops” are either fetches from L2, or flops that cannot be issued until such a fetch completes.
- “MM flops and memops” are either fetches from main memory, or flops that cannot be issued until such a fetch completes.
- “Barrier MPI” are MPI global barriers for synchronization (minuscule contribution for WRF).
- “Message MPI” are MPI messages involving data exchange.

Note that the “easy” + “other” pie slice is approximately the same size as the L1 and MM slices while L2 is slightly smaller. An almost exactly balanced machine for this problem would result if DataStar’s L2 latency were a little longer (approximately 50 cycle) and then the L2 slice would be about the same size as the others. Also note that WRF is not spending a high percentage of its time doing MPI. In general, we would not consider a machine or application that is spending a large fraction of its time doing communications, or one that would benefit as much from improving communications as from improving the processor and memory, as “balanced” because we think of communications as being pure overhead (that is, something to be minimized). And usually, for

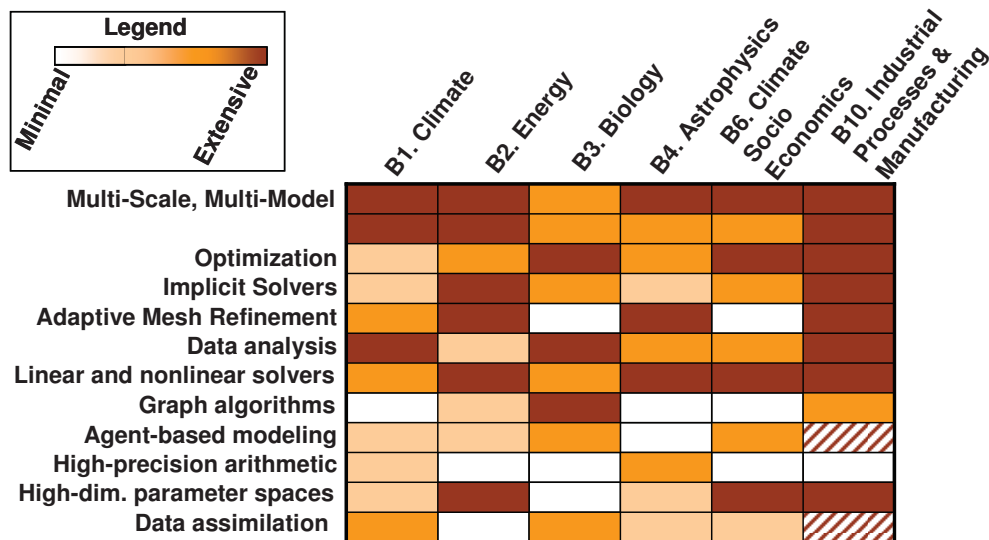


Figure 5.3: Application functionalities.

a parallel application such as WRF, if communications time is the same or greater than compute time in the other categories, then the application is past the knee of its scaling curve and should just be run on fewer processors (this may be not possible if the problem is too large to fit in the physical memory of a reduced number of processors).

By Amdahl's Law (which limits the rate of return for speeding up just one part of a multi-part task), Moore's Law, even if it continues to hold, will not speed up memory-bound applications such as WRF by more than a few percent. For example the MM flops and memops section of the pie chart would not shrink, unless new technologies are applied to improve or mitigate red shift. If doubling of transistor density per chip is just devoted to speeding up mathematical operations that are already 100 times faster than argument fetching and storing in code that (like the example) has about an equal mix of both, then not much performance improvement of memory intensive applications will be gained.

In either case it is clear that widening, mitigating, or eliminating the Von Neumann Bottleneck must be a thrust of research to enable Exascale computing as it lies in the path to increasing calculation speed by 1000x.

5.4 Exascale Application Characteristics

Exascale applications, particularly for the departmental and data center classes of problems, are liable to be rather complex in their structure, with no single overriding attribute that governs their characteristics. To get some insight into this, Figure 5.3¹ looks into several classes of applications taken from [54], and what kinds of lower level functionalities would be found in them. In this figure, the columns represent classes of applications and the rows represent different types of algorithms that might be employed in one form or another. The degree of shading in each box represents the degree to which such algorithms play an important part of the application.

¹Figure courtesy of D. Koester from [84]

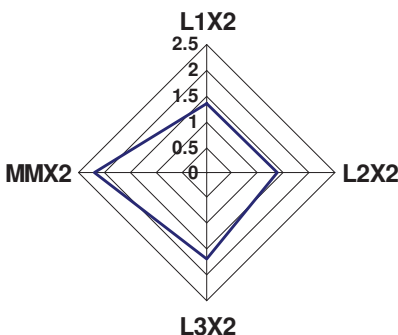


Figure 5.4: Predicted speedup of AVUS to latency halving.

The key take-away from this is that the days of very regular, simplistic, applications are over, and that applications of the future will involve a richer and more diverse suite of underlying algorithms, all of which must perform well on an underlying machine architecture in order for a system to deliver truly significant and efficient performance.

The following sections delve in more detail into some of these applications and algorithms.

5.5 Memory Intensive Applications of Today

Of all the performance parameters discussed earlier, memory latency in all its forms is often the dominant one. Applications for which this is particularly true are called **memory intensive**, and are discussed here.

5.5.1 Latency-Sensitive Applications

Figure 5.4 through 5.7 depict Kiviat diagrams predicting the performance speedup of several strategic applications due to a halving of latency to different steps in the target machine's memory hierarchy (thus shrinking the length of the bottleneck). These applications cover a spectrum of strategic science uses:

- **AVUS** (Air Vehicle Unstructured Solver) is a CFD code used in airplane design,
- **WRF** is a weather prediction code mentioned above,
- **AMR** (**A**daptive **M**esh **R**efinement) is a benchmark representative of many domains of computational science,
- **Hycom** is an ocean modeling code.

These applications or their variants are run in a production mode by federal agencies including DoD, DoE, NSF, and NOAA, and consume tens of millions of supercomputer hours annually. They are not chosen for this study solely because they are memory intensive but because they are scientifically and strategically important.

The speedups graphed above in Figure 5.4 through 5.7 are forecast by performance models relative to an existing base system (a 2048 processor IBM Power4 based supercomputer). Each radial axis of these graphs represents the predicted speedup of halving the latency (in Power4

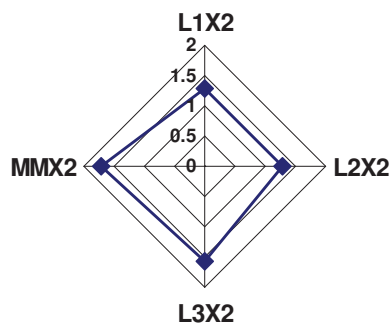


Figure 5.5: Predicted speedup of WRF to latency halving.

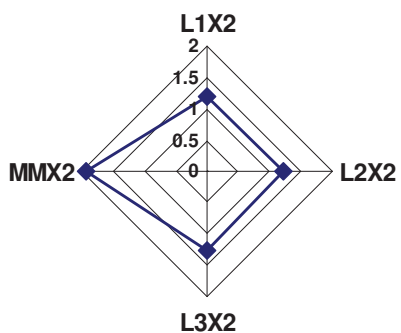


Figure 5.6: Predicted speedup of AMR to latency halving.

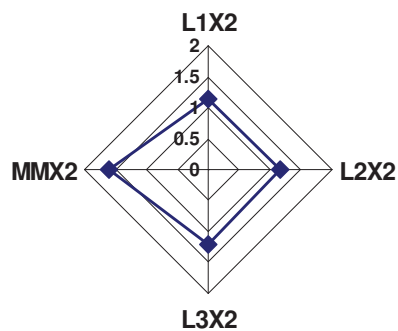


Figure 5.7: Predicted speedup of Hycom to latency halving.

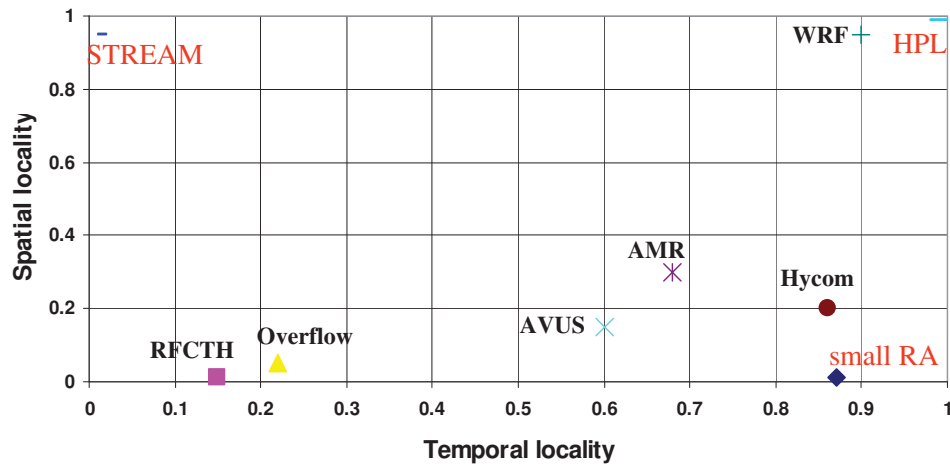


Figure 5.8: Spatial and temporal locality of strategic applications.

cycles) of memory accesses to successive levels of the memory hierarchy (“L1” is the level 1 cache, “L2” is the level 2 cache, etc. “MM” is main memory). The units of each axis are predicted speedup of the application’s total run-time relative the base system’s speed if the base machine were to be improved in just that dimension. For example, a value of 1 on an axis would represent no predicted speedup of the application for halving the latency to that level of the memory hierarchy while a value of 2 would predict the most speedup one could reasonably expect from halving the latency of that level of the memory hierarchy (and imply the application is only doing memory references to that level of the memory hierarchy).

Of note is that all these applications are forecast to benefit more from decreasing main memory latency than from decreasing L1 (on chip) latency. So Moore’s law, even if it continues to hold, will not help these applications much unless it is exploited in a way different from the ways it has been historically.

5.5.2 Locality Sensitive Applications

The performance of memory intensive applications such as the above on today’s machines depends mostly on the application’s spatial and temporal locality and the ability of the machine to take advantage of those to mitigate the effect of the von Neumann Bottleneck. Here by **locality** we mean the likelihood of a memory reference will be in some sense “local” to some prior memory access. **Spatial locality** is the tendency of a computation to access memory locations that are contiguous by address location to prior references - these addresses are then amenable to prefetching which can improve performance by hiding latency when they are accessed. **Temporal locality** is the tendency of a computation to access memory locations that it has already accessed - these addresses are amenable to caching which can improve performance by storing the data in small, near-to-the-processor memories which can be accessed with reduced latency.

Figure 5.8 shows spatial and temporal locality of a number of strategic applications (including those of Figures 5.4 2 through 5.7) where spatial and temporal locality are assigned a numeric score in the range [0,1] as defined in [155], with a score of 1 being the highest possible locality (every reference is local to some previous one) and a score of 0 being the least (there is no locality correlation of any kind).

To help put these applications into perspective with respect to locality, they are plotted in Figure 5.9 along with a suite of smaller kernels (highlighted):

- **HPL**, a small-footprint, cache-friendly benchmark with near maximal temporal locality used to generate the Top500 list,
- **STREAM**, a unit-stride benchmark with near perfect spatial locality but no reuse,
- **small Random Access (RA)**, a benchmark with almost no locality whatever.

HPL, for comparison, runs very efficiently on modern machines where it gets “balanced” performance, that is, it usually achieves about 1 cycle for a memory reference to on-chip L1 and thus can run at near theoretical peak floating-point issue rate.

STREAM gets reasonably good performance on modern machines as it uses a whole cache line and benefits from prefetching; when a large-memory STREAM test case is run its performance is limited by bandwidth (rather than latency) to main memory.

Small Random Access performs poorly on most modern machines, at about the latency of main memory (in the neighborhood of greater than 100 cycles per memory operation). This is despite the fact that it fits in cache like HPL, but jumps around without striding through cache lines.

Real applications fall between these kernel extremes both in locality and in performance as shown in Figure 5.9 where performance in terms of something proportional to “cycles per instruction” (CPI)/ondex(CPI) is plotted on the Z axis. Higher numbers on this Z axis correspond to “lower performance.” As can be seen from this Figure, applications with higher locality are “easier” for today’s machines to run fast; they average lower cycles per operation by either doing more memory references in nearer caches (reducing latency) or prefetching data from future addresses (mitigating latency by Little’s Law).

Unfortunately, however, there are strategic applications that are even more memory intensive than those graphed in Figures 5.4 through 5.7 such as **Overflow**, a CFD code run by NASA, and **RFCTH**, a blast physics code. These perform but poorly on all of today’s machines.

As a general rule of thumb: these applications require a minimum of 0.5GB of local main memory for each 1 GFlops of processor (and some such as Overflow and RFCTH require more than 2x that).

Also as a general rule the amount of level 2 and level 3 cache required by these applications corresponds inversely to their temporal locality score (see reference for details). For example WRF is relatively cache-friendly code and can get by today with an L2 cache size of 1MB while Overflow and RFCTH need at least a 12 MB L3 cache (and would perform better if it were a 12 MB L2).

5.5.3 Communication Costs - Bisection Bandwidth

Generally speaking as depicted in Figure 5.10, the fraction of time spent in communications increases for these applications as a fixed problem size is solved with more processors. In such cases, increasing the number of processors results in smaller amounts of data that is “close to” each processor. In other words an increase in concurrency is offset by a decrease in locality; and at some point diminishing returns are reached.

A current standard metric for such communication costs is **bisection bandwidth**- if a system is arbitrarily divided into two halves, the bisection bandwidth is the minimum bandwidth that is still guaranteed to be available between these two halves. How well this metric holds up in the future will depend on the characteristics of the applications and the patterns of how different sites of computation must communicate with other sites, such as:

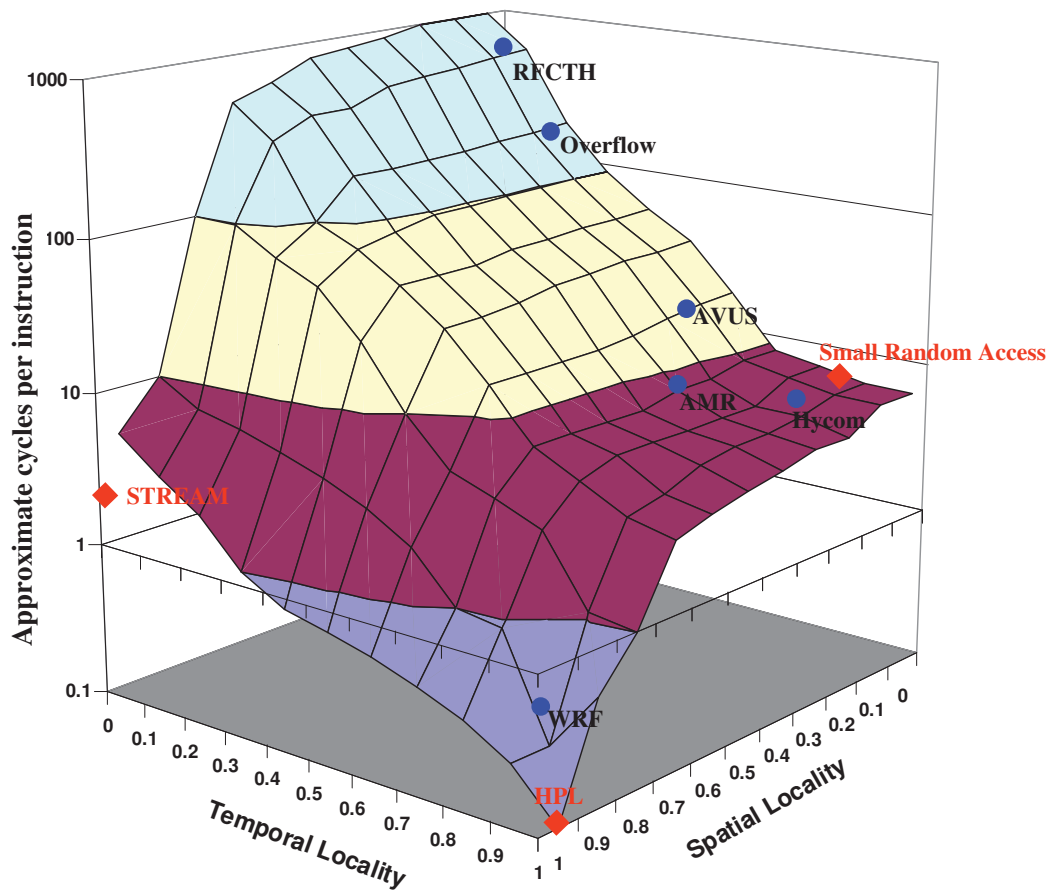


Figure 5.9: Performance strategic applications as a function of locality.

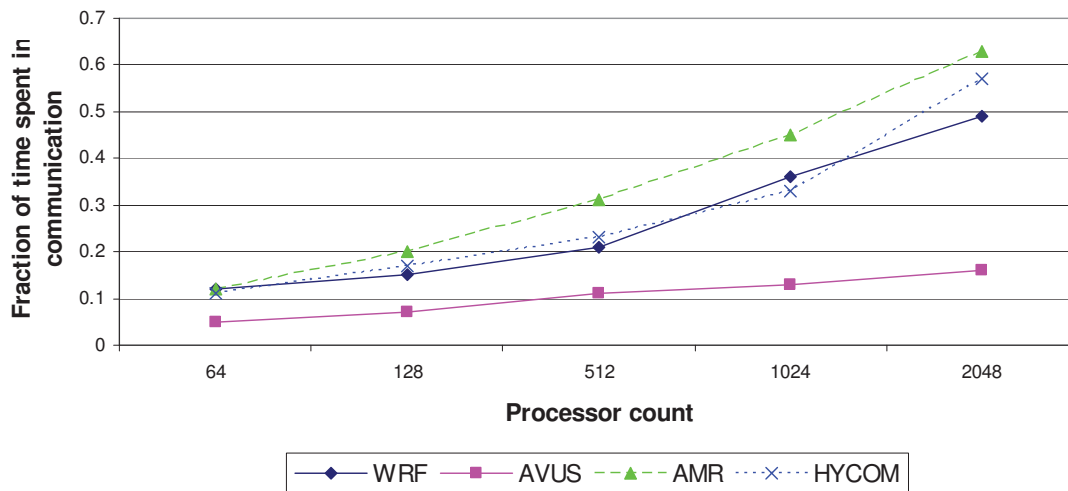


Figure 5.10: Growth of communications overhead.

- scientific and engineering codes such as PDEs and 3D meshes:
 - structured grids involve nearest neighbor communications,
 - unstructured grids where indirection through link tables is the norm,
 - adaptive mesh refinements where a fair amount of data must be moved around the machine at unpredictable times to rebalance and load-balance the application.
- multi-scale and multi-model applications that act like amalgams of multiple “single algorithm” applications as above, but must do so simultaneously, meaning that any sort of regularize mapping between data and processors becomes very difficult, and the aggregate bandwidth patterns become “random.”
- new applications such as those involving large unstructured graphs may simply be impossible to partition to minimize communication for long periods of time.
- as discussed in Section 6.7.4, the growing need to “copy memory” for application-level data checkpointing to increase the resiliency of the system to internal faults.

5.6 Exascale Applications Scaling

The applications discussed in the previous section require around 0.5GB per Gflops of computation when run on today’s high end systems. How such numbers, and similar parameters for other applications, will change as we move towards Exascale is crucial to sizing the complexity and basic architecture of Exascale computers, and is the subject of this section.

5.6.1 Application Categories

To start such discussions, we consider four categories of applications in terms of their “scalability,” and ordered in approximate ease of implementation:

- I. Those applications that would solve the same problem as today but with a 1000x more data points. This represents using the “same physics” and the “same algorithm,” but over larger surfaces, and is normally called “weak scaling.” In this category might be, for example, global weather models at sub 1km resolution (effectively solving a WRF calculation at the granularity used today to forecast hurricanes but at the breadth of the entire earth’s atmosphere rather than over the Gulf alone).
- II. Those that could solve the same problem as today but 1000x faster. In this category for example might be real-time CFD to stabilize a physically morphable airplane wing (effectively recalculating every few seconds an AVUS calculation that now takes hours). Another example would be running an operational weather forecast of a hurricane storm track in minutes rather than days (thus improving advanced warning).
- III. Those would that would solve the same problem, at the same size, as today, but with 1000x more time steps. In this category for example might be local weather models at climatic timescales (effectively solving a WRF calculation at the size today used for medium-term weather forecasting but projecting out for centuries).

- IV. Those that would solve the same problem as today but at 1000x more resolution (usually requiring increased physics and chemistry in every cell). In this category might be for example global ocean and tide models such as Hycom to include micro-features such as wave refraction and coastal turbulent mixing.

Category I applications are those that often go by the term “embarrassingly parallel” where it is obvious how to partition instances up into almost arbitrarily many pieces that can all be computed in parallel. Thus systems with more parallelism in them can in fact be used by such applications for near linear increases in performance.

Categories II, III, and IV are all versions of **strong scaling**, with category II being the conventional version, but category IV being the most challenging because of the need to model more science with more computation. In all of these categories, the relationship between parallelism and performance is highly non-linear.

Clearly Exascale applications could have aspects of any combination of the above; for example one could perform a calculation at 100x resolution and 10x more time-steps, for a total of 1000x more computation than is possible today.

5.6.2 Memory Requirements

We next want to consider three aspects of the **memory footprint** of Exascale applications:

1. their total memory size requirements,
2. the size of working sets (**locality clusters**) in their data that in turn would determine the sizes of local main memory and local caches they require, and
3. the latency and bandwidth requirements they would place on each level of the memory hierarchy, including global interconnect.

As to applications in category I above, the total memory footprint would increase 1000x but the sizes and latencies of the local memory hierarchy would not need to change relative to today. However, to enable scalability and efficient deployment of more processors, the interconnect between processors would have to improve in latency, or trade latency for bandwidth, in proportion to some function of the topology (i.e. square root of 1000 \approx 32-fold for a torus) to preserve the performance of global data communications and synchronization.

As to category II above, the total memory footprint would not change. But latency would have to be improved 1000x to each level of the memory hierarchy unless a combination of code tuning and machine features could unlock sufficient parallelism in memory references to cover some of the additional latency requirements via increased bandwidth². In that case it would be the bandwidth that would need to be improved proportionally.

Category III requirements are the same as for category II.

Category IV applications are similar to category I unless the higher resolution problem is highly parallelizable (and this depends on the nature of the science problem on a case-by-case basis) in which case it resembles more category II. In other words, if additional resolution or additional physics in the local calculation does not improve coarse-grained task parallelism, then improved latency (or improved local memory bandwidth and ILP) is the only way to increase memory performance. If on the other hand, additional resolution or physics results in greater task parallelism,

²Bandwidth can be traded for latency by Little's Law

then one can achieve increased performance by adding more processors that each look like today's, a la category II.

Thus we have four categories of potential Exascale applications, and several dimensions of Exascale memory technology to consider that could speed them up 1000x. Futuristic AVUS is perhaps an exemplar of Category II, variants of WRF could be category I, II or III, and extensions to Hycom of category IV.

5.6.3 Increasing Non-Main Memory Storage Capacity

Historically, storage beyond the DRAM of main memory has fallen into several categories:

- **scratch storage** used for both checkpointing and for intermediate data sets needed for “out of core” solvers.
- **file storage** used for named input and output files that will persist beyond the lifetime of the application execution.
- **archival storage** used for long term storage of data.

5.6.3.1 Scratch Storage

Scratch storage in the past has been primarily driven by the size of main memory and the need to periodically checkpoint (as discussed in Section 6.7.4. To date its implementation has been as a large number of disk drives.

In the future, however, there will be an increasing need for larger scratch storage uses, such as for applications that dynamically construct data derived models, that have such large memory needs that “out of core” algorithms are necessary, and in capturing performance monitoring data that is used by the system to do dynamic load-balancing and performance tuning.

Thus, it should be expected that such storage needs will grow on the order of 10 to 100X the size of main memory.

5.6.3.2 File Storage

File storage represents persistent data sets that have real value to the end users (either as input files or summary outputs), and may be shared among different applications that run at different times. In addition, as systems become larger, at least a subset of the performance monitoring data mentioned in the scratch storage discussion may need to be saved, along with fault data, to look for trends that may require either repartitioning of future application runs or scheduling down time for maintenance.

Besides the capacity issue, there is also a significant **metadata** problem with file storage, where metadata relates to the directory information needed to both locate individual files in the system, and monitor the state of those files. Today, there are many applications that even on sub petaflops machines keep literally hundreds of thousands of files open at a time, and that number is growing.

At the end of the day, file system size is thus also at least linear with main memory, with perhaps a 1000X multiplier present for Petascale systems.

5.6.3.3 Archival Storage

The size of archival storage is highly dependent on the application and the problem size. Traditionally, it has been driven by both checkpoint/restart as above, and by requirements where

subsets of such “checkpoints” represent time points in the computation that are to be used in 3D visualizations. The latter terms will certainly grow rapidly as Category III applications become important.

Further, as bad as the metadata problem is for file storage, it is even worse for archival storage, especially as the number of time steps per application run increases.

Even within current supercomputers where the computational assets are unchanging, the growth in archival storage needs has been significant, with 1.7 to 1.9 CAGR observed.

A recent study[55] about capturing data from NASA’s **EOS/DIS** (Earth Observing System/-Data Information System) not only analyzer the cost efficiency of several approaches before suggesting tape farms, but also introduced two additional classes of metrics: the number of (kilo-/mega/giga) objects servable from such a system per second, and (more interestingly) the number of “scans” of the entire data set that can be done per day. This latter term becomes critically important as data mining and similar utilities become important.

In summary, such needs signal storage requirements that are easily in the 100X main memory category.

5.6.4 Increasing Memory Bandwidth

Increasing memory bandwidth to the local processor would most benefit Category II, and Category III applications. A machine that can do this would have to have radical technology advances to improve local memory bandwidth such as 3D stacked memory chips with higher speed signalling.

5.6.5 Increasing Bisection Bandwidth

Increasing bisection bandwidth becomes very expensive as the system grows in size, and may be a strong function of the maximum scale of the system and target application class. For example, an Exascale departmental system that will execute variations of today’s applications (i.e. largely Category I and II) should probably have a bisection bandwidth comparable to that provided by the current 3D system topologies, but scaled to a petaflops. For example, a current XT4 supporting a 40x32x24 topology at 318 Tflops has a bisection bandwidth of 19.4TB/s. Simply scaling each dimension by a factor of 2 (to 80x64x48), and scaling the flops to 2.4 Pflops would thus require about 80 TB/s. For other Petascale applications, bisection bandwidth may be approaching the HPCS goals of 0.5 to 3.2 PB/s. Thus, overall, machines will become useful with bisection bandwidths in the O(50 TB/s) to O(1 PB/s) range.

As will be discussed later, Exascale applications that are destined for execution on the data center class systems may scale differently. While an “existence proof” application (Category I) may be runnable on systems with bandwidths in the O(1 PB/s) range, more realistic applications may have to grow with the memory bandwidth (reflecting more random data exchange patterns), and thus soar into the O(10 PB/s) to O(1 EB/s) range.

5.6.6 Increasing Processor Count

Increasing the number of processors (and local memory subsystems) would most benefit applications in category I. A machine that can do this would have to have radical advances in scalability with perhaps hundreds of millions of processors.

Category IV may require a combination of improved local memory bandwidth and increased number of processors and local memories.

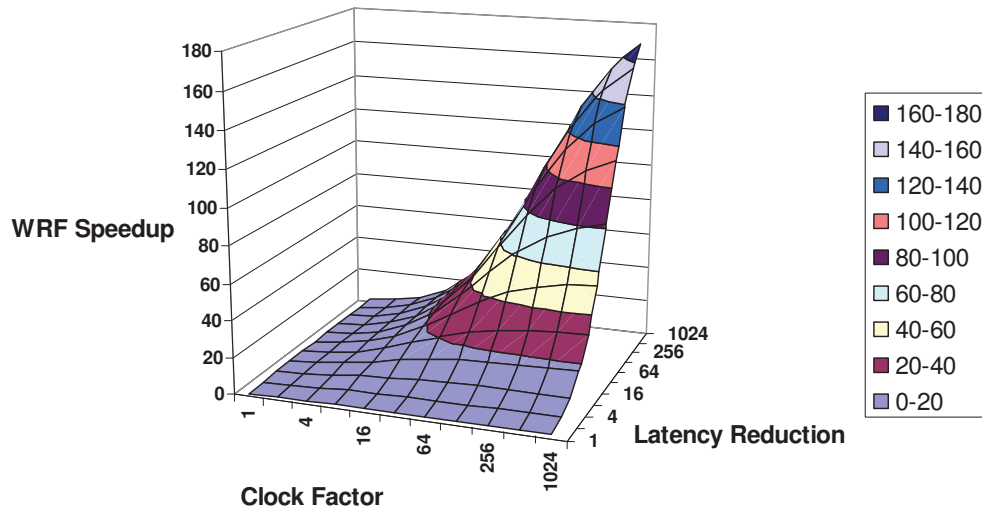


Figure 5.11: WRF performance response.

5.7 Application Concurrency Growth and Scalability

Quantifying what it means to be ‘1000x harder’ has aspects in:

- Performance Growth
- Storage Growth
- Interconnect Bandwidth Growth
- Scratch Storage Growth
- File System Growth

All of these will grow more or less than 1000x, often independently. This section looks at the process of making such extrapolations, looking in particular for non-linear characteristics.

5.7.1 Projections Based on Current Implementations

For projecting to Exascale, we focus on three applications from Figure 5.9 that could be considered “Easy,” “Hard,” and “Harder” from a memory spatial and temporal locality perspective: HPL, WRF, and AVUS. We note that in the nomenclature defined above, WRF is a Category I, AVUS a Category II application, and WRF is potentially a Category III application. For these projections we ignore Category IV because its requirements for memory and number of processors fall somewhere in between Category I and Category II.

Figures 5.11 through 5.13 depict the performance response surface as predicted by the Convolution Method[130][131], and give forecast speedup for WRF, AVUS, and HPL as a function of increasing on-chip operations rates such as flops (X axis), or decreasing latency to memory at L2 and main memory level inclusive (Y axis). These convolutions are hardware agnostic, and do not specify how such improvements would be obtained (as for example whether by adding more

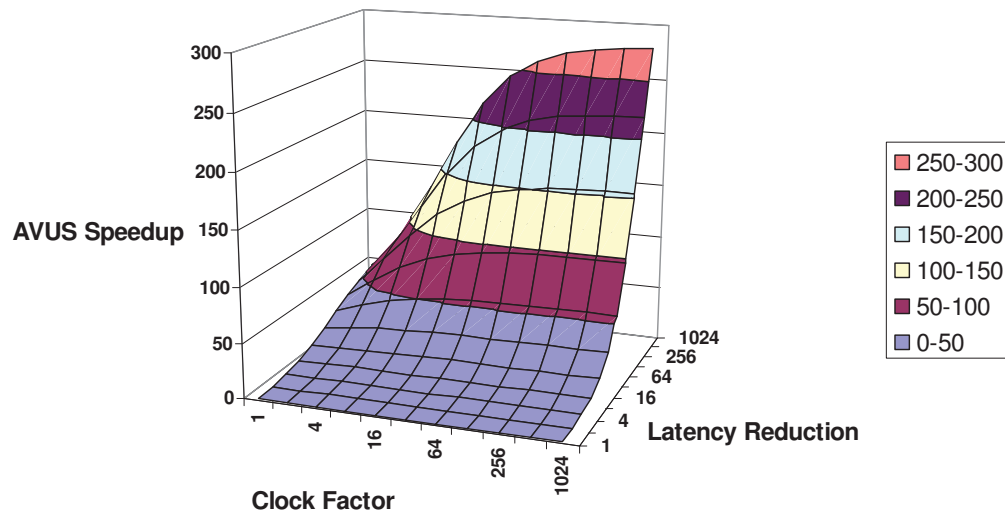


Figure 5.12: AVUS performance response.

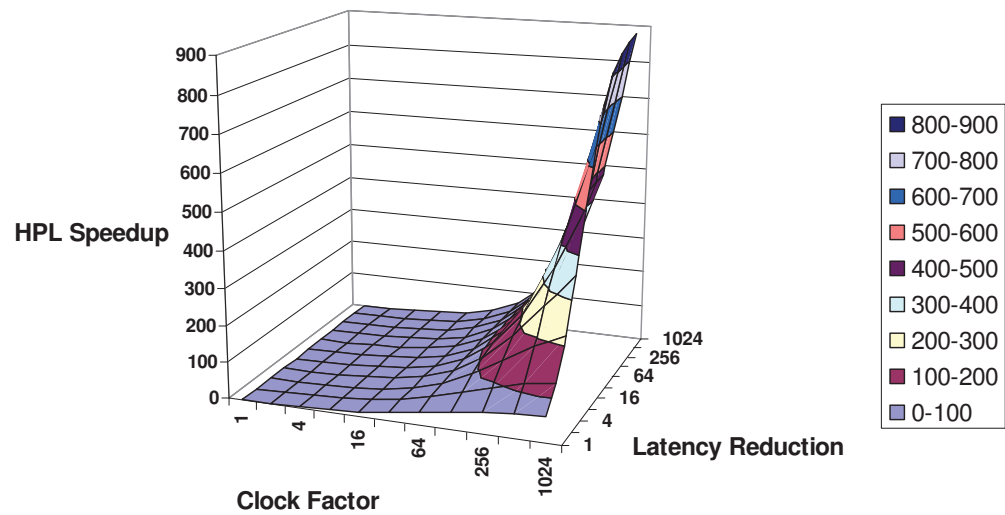


Figure 5.13: HPL performance response.

processors or by speeding up the clocking rate or otherwise improving the processors and/or memory). They simply assume that the machine has some way of doing work in the above operation categories faster.

However, an underlying assumption in Figures 5.11 through 5.13 is that communications overhead is a constant. (This perhaps overly optimistic assumption will be restricted later below). Note even so, for example in Figure 5.2, that an initially small fraction of communications becomes the bottleneck ultimately (by Amdahl's Law); thus even with 1024-fold improvement of both flops and memory, WRF can only be sped up about 180-fold according to the predictions. If communications latencies also could be improved by the same factors, then the WRF performance response would still have the same shape as Figure 5.2 but would reach higher levels of speedup.

Further, if one can trade bandwidth for latency then one could relabel the Y axis of Figure 5.2 and the subsequent figures as "memory bandwidth" rather than "1/memory latency." However the ability to trade bandwidth for latency tolerance via Little's Law is a complex function of the machine and the application. One needs to know for example, "how much inherent instruction-level parallelism (ILP) is in the application?," and "how much hardware support for in-flight instructions is in the hardware?." This is a level of detail not incorporated in the convolutions shown here. Thus Figure 5.2 and subsequent figures could be considered the most optimistic performance results obtainable by improving memory bandwidth rather than latency under the assumption that one could use all that extra bandwidth to tolerate latency better (and real performance would probably be less for that approach). Convolutions could of course be carried out under more pessimistic assumptions, but that is reserved for future work.

The "balanced" nature of WRF is apparent again from Figure 5.11. The performance response is symmetric for improving flops or memory latency. Once one is improved the other becomes the bottleneck.

By contrast, comparing Figure 5.12 for AVUS to Figure 5.11 for WRF above, we see that AVUS is a much more memory intensive code. Indeed, one must improve memory latency (or perhaps bandwidth, see discussion above) by an order-of-magnitude to unlock measurable improvements due to improving flops rate. One could even improve flops three orders of magnitude and get less than 50-fold AVUS speedup if the memory subsystem is not also sped up. On the other hand, AVUS does even less communications than WRF and so speeds up better for the same memory improvements - high-bandwidth processors would help AVUS, and other such memory intensive applications, a great deal.

Comparing Figure 5.13 for HPL to Figures 5.11 and 5.12 demonstrates what a pathologically useless benchmark HPL is as a representative for broader classes of applications. Its peak is overly optimistic (as it has almost no communications) and its lack of memory latency or bandwidth demands is not representative of more robust calculations such as WRF and AVUS. Never-the-less, even HPL will only speed up two orders of magnitude for three orders of magnitude improvement in flops if some improvement of memory latency (or possibly bandwidth) is not also accomplished.

Figures 5.11 through 5.13 then are most straightforwardly interpreted as applying to fixed problem size and processor/memory count, where it is reasonable to assume that communications does not grow in an absolute sense, but just as a fraction of total run-time due to Amdahl's Law.

Now let us consider the case of weak scaling, that is, building a system out to more processors and memories to enable more aggregate flops and bandwidth, and also making the problem bigger. Both of these approaches are not unreasonable to consider for WRF and AVUS as there are indeed larger problems (whole Earth atmosphere, full jet plane under maneuver) that can approach Exascale and still be scientifically interesting. Figures 5.11 through 5.13 still make sense if we interpret "speedup" to mean doing more work in a fixed time. But again recall the assumption that communications time is a constant. Once again, for this to be true, it would seem to imply

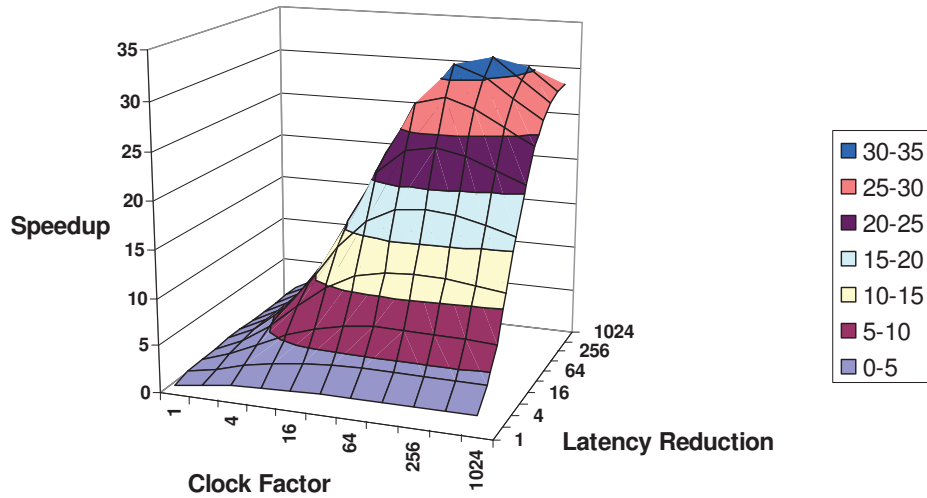


Figure 5.14: WRF with $\log(n)$ communications growth.

we can trade increased bandwidth of the larger machine to cover increasing latency. Whether this is true would again depend on the application and the architecture.

Figures 5.14 and 5.15 then give more pessimistic performance response surfaces of WRF and AVUS if communications cost instead grows as a logarithmic function of improving flops or memory latency/bandwidth (i.e. increasing problem size in the weak-scaling approach). One would have to improve network latencies, or trade bandwidth for latency, at a rate better than $\log(\text{cpu count})$ to be able to get more than about 1 order of magnitude performance improvement reasonable for scaling out by 1000X one of today's machines 1000X.

Figures 5.14 and 5.15 then seem to argue for the importance of improving processor and memory performance, especially to benefit Category II and higher, not just scaling out huge systems for Category I, to enable a broad spectrum of important applications to operate at Exascale.

5.7.2 Projections Based on Theoretical Algorithm Analysis

A Petascale calculation of today, such as for the WRF projections in the previous section, is an operational hurricane forecast. It requires both ultra-high-resolution of gradients across the eye-wall boundaries (at ≈ 1 km or less), and representation of the turbulent mixing process correctly (at ≈ 10 m or less). Today's Petascale hurricane forecasting computation might have the following parameters:

- a 100 kilometer square outer-most domain at 10 meter horizontal grid spacing and 150 vertical levels,
- a 15-billion cell inner-most 10 meter nested domain,
- with a model time step of 60 milliseconds

Such a computation consumes about 18 machine hours per simulated day at a sustained petaflop/second on 100,000 processors and takes up about 100 MB per task of data not counting buffers, executable size, OS tax etc. (10 TB of main memory for application in aggregate). The computation generates 24 1.8 terabyte data sets, or 43.2 TB per simulation day if hourly output of

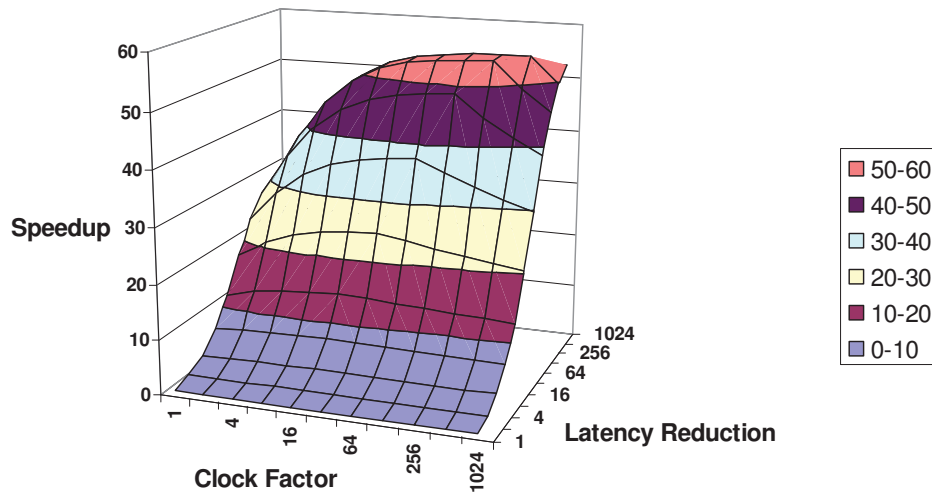


Figure 5.15: AVUS with $\log(n)$ communications growth.

30 three-dimensional fields is performed. At an integration rate of 18 machine hours per simulated day at a sustained petaflop, the average sustained output bandwidth required is 700 MB/second.

If we project a Category II Exascale WRF calculation (with a goal to track a single storm 1000X faster) then no increase in physical memory size or I/O bandwidth is needed (though floating-point issue and memory bandwidth have to improve by 1000X). This would allow forecasts of storm tracks with lead time of days or even weeks instead of only hours or minutes, and would greatly improve disaster preparedness lead time.

A different Petascale calculation of today is a WRF “nature run” that provide very high-resolution “truth” against which more coarse simulations or perturbation runs may be compared for purposes of studying predictability, stochastic parameterization, and fundamental dynamics. Modeled is an idealized high resolution rotating fluid on the hemisphere to investigate scales that span the k-3 to k-5/3 kinetic energy spectral transition. Computational requirements for a 907x907 grid with 101 levels, resolution at 25km, time step at 30s on 100,000 processors, is 100 MB main memory per task (= 10 TB of main memory) and outputs about 100 TB per simulation day, or a sustained 1K MB/s I/O.

Another Category I Exascale WRF calculation would be to perform a full nature run of 1 hemisphere of earth at sub 1km resolution. This would capture small features and “the butterfly effect” - large perturbations at long distance driven by small local effects. This very challenging calculation at a sustained Exaflop would then require 10,000 GB = 10 PB of main memory. I/O requirements would also go up 1000x.

If we project a Category III Exascale WRF hurricane calculation (track 10 storms of interest at once as part of a simulation of a portion of the hurricane season), and predict each storm track 100X faster, then memory requirements scale up 10X and I/O 10X.

In other domains of science such as mantle physics, modeling earthquakes with multiple scenarios in short term disaster response or evolution of fault on geological timescale, is a “capacity” (Category I) problem, while modeling the mantle of earth as a “living thing” coupled to crust for understanding tectonic plate system evolution is a “capability” (Category II) problem.

In biology, protein folding very long sequences, interactive protein docking, and calculating multiple drug interactions is a “capacity” (Category I) problem, while modeling membranes, organs,

organisms, and even going the other direction to cell modeling at molecular level of detail is a "capability" Category II problem.

5.7.3 Scaling to Departmental or Embedded Systems

Pioneering Exascale applications, like those first expected to run at Petascale in the next year, will likely be a handful of specialized scientific codes. These applications will be very narrowly designed to solve specific problems. They will be selected for their ability to scale to billions of concurrent operations without having to solve currently intractable problems such as partitioning unstructured grids. They might relax normal synchronization constraints in order to avoid the resulting bottlenecks. With heroic programming efforts, they will execute out of the smallest, fastest levels of the memory hierarchy, minimizing the number of concurrent operations needed to cover main memory latency.

For there to be a viable market for Petascale departmental systems, there will have to be viable commercial applications to run on them. Such Petascale commercial applications will need to sustain millions of concurrent operations, a daunting task when today, only a handful of commercial codes can sustain even $O(1000)$ concurrent operations on a few hundred multi-issue CPUs. Thus the developers of such codes will share many of the same challenges as those developing Exascale applications for the nation, how does one add four orders-of-magnitude to the level of concurrency one struggles to sustain today. Other issues such as debugging at this scale or tolerating errors in increasingly unreliable systems will also be common to developers of both Exascale and departmental scale programs.

There will also be significant differences in the challenges facing commercial Petascale software. Rather than solve a handful of problems of national interest, they will have to solve a broad range of engineering and business problems. Rather than be small scale versions of Exascale scientific codes, they will be Petascale versions of codes that also run on Terascale desk top systems. As such, they will enjoy large numbers of users and hence be economically viable. However, their developers will have to address a broader range of problems, including unstructured grids and more robust numerical algorithms. Such applications today are millions of lines of code, and represent an investment in labor that cannot be repeated, but rather must evolve into the future. Thus any new programming languages or features must be backward compatible to be accepted into commercial software.

Petascale application will also put demands on departmental systems that make them different than simply smaller Exascale systems. Petascale departmental systems will have to provide full featured operating systems, not micro-kernels. They will have to efficiently process large ensembles of end-user applications, not merely a handful of heroic jobs. Where Petascale scale systems do not have enough main memory, their applications will go "out-of-core," requiring a disproportionately larger and higher performing file system to hold the balance of their state.

Scaling to embedded systems will also be an issue for Exascale applications. Just as today one sees applications such as sPPM running on everything from the largest supercomputers (BG/L) to Playstations, one can expect the same in the next decade, as users try to exploit any computing system available to them. There will be additional challenges however, making embedded applications more challenging than their departmental counterparts. Embedded codes often involve similar computational kernels, but they tend to be implemented with specialized software. This reflects both physical constraints such as volume and power, which reduce the size and performance of the processors and memory, as well as novel software environments, such as real-time operating systems with less functionality than those expected on departmental servers.

		Departmental Class		Data Center Class	
		Range	“Sweet Spot”	Range	“Sweet Spot”
Memory Footprint					
System Mem- ory		O(100TB) to O(1PB)	500 TB	O(1PB) to O(1EB)	50 PB
Scratch Stor- age		O(1PB) to O(100PB)	10 PB	O(100PB) to O(100EB)	2 EB
Archival Stor- age		>O(100PB) to O(100PB)	100 PB	>O(100EB)	100 EB
Communications Footprint					
Local Memory Bandwidth and Latency		Expect low spatial locality			
Global Mem- ory Bisection Bandwidth		O(50TB/S) to O(1PB/s)	1PB/s	O(10PB/s) to O(1EB/s)	200PB/s
Global Mem- ory Latency		Expect limited locality			
Storage Band- width		Will grow at faster rate than system peak performance or system memory growth			

Table 5.1: Summary applications characteristics.

5.8 Applications Assessments

5.8.1 Summary Observations

In terms of overall system characteristics, Table 5.1 attempts to summarize a first cut at how the different memory, storage, and bandwidth considerations might play out for both a Departmental class Exascale system and a Data Center class. It should be stressed that these numbers are relatively speculative right now, and can only be validated when extensive work on real Exascale applications is performed.

For a bit more detail on the internal structure of Exascale applications and how they may achieve Exascale performance, Figure 5.16 attempts to summarize some of the trends observed in terms of applications over time. The vertical axis refers to locality - how much information about one memory reference can be used to improve the latency of accessing a future one. The horizontal axis refers to how much concurrency is present.

The contents of the graph refer to classes of applications. Looking backwards in time, many early “high end” numeric applications used simple 3D regular grids where the data could be positioned precisely before hand, and techniques such as red-black ordering provided potential levels of parallelism that approached the number of grid points.

Moving forward to today, grids have become much more irregular, with a great deal of auxiliary table look-ups needed to account for detailed modeling. The result has been some more potential for concurrency (more points), but significantly decreased locality. This locality becomes even worse when dynamic mesh refinement is used to change the grid dynamically.

Another class of applications are highly non-numeric, and have significantly different characteristics, particularly in locality, than the first class. Searching a simple one-dimensional linked list, for example, has almost no possible concurrency if no pre-computed pointers to intermediate nodes

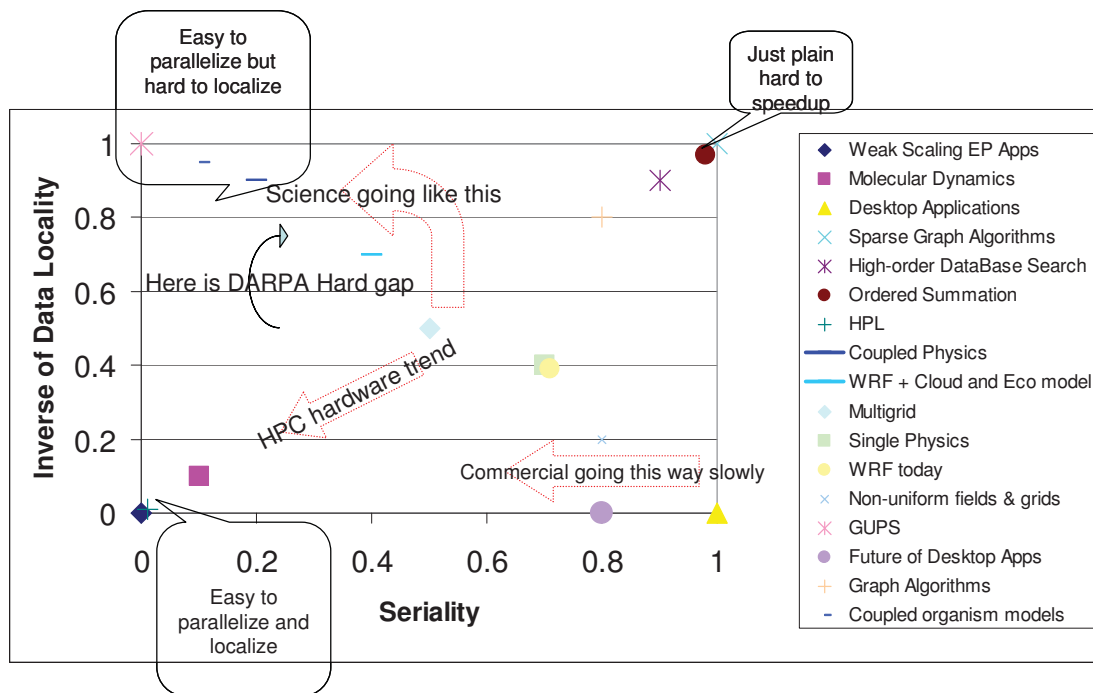


Figure 5.16: Future scaling trends

exist to allow searches to jump into the middle of the list. It also has potentially very low locality if each node of the list ends up in different memories.

Going forward to more relevant applications, searching large distributed graphs has as little locality as the 1D search, but with a parallelism limited by the diameter of the graph. Finally, applications similar to GUPS have potentially huge amounts of parallelism, but again little or no locality.

5.8.2 Implications for Future Research

The discussions in this chapter lead fairly directly to several research directions that would directly impact the ability of applications to take advantage of Exascale hardware:

- Develop additional hardware and architectural techniques to mitigate the impact of poor locality within an application.
- Provide more hardware and architectural hooks to control the memory hierarchy, and provide programming metaphors and APIs that allow an application to express how it wants to control locality of data within this hierarchy.
- Through compiler technology, program transformation techniques, or address reordering techniques, find or create improved locality automatically.
- For applications of direct interest to DoD, foster algorithm work that focuses on both parallelism and locality.

- Develop algorithmic and programming techniques that can tolerate poor locality, such as increasing asynchronicity in communications, and more prefetch opportunities.

In addition, in support of nearly all of these research directions, it appears reasonable to develop suites of tools that can provide estimates of “upside potentials” of current and emerging codes, including:

- tools to analyze existing codes for dependencies and sequences of memory access addresses,
- tools to “data mine” the outputs of the above to look for and identify “patterns,”
- tools that allow such patterns to be translated into forms that can be used by systems to implement these patterns in ways that increase performance,
- tools that analyze the codes to provide estimates of “oracle” parallelism opportunities, and transform that into estimates of scaling when run on parallel hardware.

This page intentionally left blank.

Chapter 6

Technology Roadmaps

This chapter reviews the suite of relevant technologies as we understand them today. This includes both those that are well-established and in current use in today's systems, and those that are just emerging. For each such technology we discuss:

- Its fundamental operation and expected target usage.
- The key metrics by which to judge its progress.
- Its current state of maturity.
- For the key metrics both current values, physical limits as we understand them today, and a roadmap as to how improvements are projected to occur with current funding and focus, with an emphasis on the time between now and 2015.
- The fundamental reliability of subsystems built from such technologies.
- Particular areas where additional research may prove valuable in accelerating progress towards reaching the limits of the technology.

The key technology areas reviewed in the sections below include:

- Section 6.2: technology from which logic and computational functions may be constructed.
- Section 6.3: technology from which the primary memory used in computers is constructed.
- Section 6.4: technology from which mass store such as file systems is constructed.
- Section 6.5: interconnect technology that permits different computing sites to communicate with a single computing structure.
- Section 6.6: technology with which combinations of chips from the above categories (especially logic and main memory) may be packaged together and cooled.
- Section 6.7: techniques used today to improve the overall resiliency of a computing system.
- Section 6.8: the (largely software) technologies for managing the operating environment of the computing systems.
- Section 6.9: software technologies for extracting the parallelism and generating the application code.

6.1 Technological Maturity

Gauging the maturity of a technology is an important aspect of performing any projections such as done in this study, especially when time frames to get to deployable systems are important. In this report we use a metric developed by NASA termed “Technology Readiness Levels,” which has multiple levels as follows (taken from [99]):

1. Basic principles observed and reported
2. Technology concept and/or application formulated
3. Analytical and experimental critical function and/or characteristic proof-of concept
4. Component and/or breadboard validation in laboratory environment
5. Component and/or breadboard validation in relevant environment
6. System/subsystem model or prototype demonstration in a relevant environment
7. System prototype demonstration in a real environment
8. Actual system completed and “flight qualified” through test and demonstration
9. Actual system “flight proven” through successful operations

Such levels address the maturity of a technology in terms of how fully it has been developed. However, it is important to distinguish such levels from the funding categories often used in describing development budgets used to bring such technologies to practice:

- 6.1 Basic Research
- 6.2 Applied Research
- 6.3 Advanced Technology Development
- 6.4 Demonstration and Validation
- 6.5 Engineering Manufacturing Development
- 6.6 Management Support
- 6.7 Operational Systems Development

It was the sense of the study group that for Exascale projects constructed from technologies currently at TRL levels 1 and 2 most closely corresponded to 6.1 projects, TRL levels 3 and 4 corresponded to 6.2, and TRL levels 5 and 6 to 6.3. Once technologies have become more mature that TRL 6, they are no longer “immature,” and need to stand on their own commercial value for continued development and deployment.

In these terms, the goal of the study is thus to determine where technologies projected to be “mature” in the 2013-2014 time frame by normal development will be inadequate to support Exascale systems, and whether or not there is the potential for supporting the accelerated development of new, currently “immature,” technologies that may bridge the gap.

	Units	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020
Feature Size	nm	90	78	68	59	52	45	40	36	32	28	25	22	20	18	16	14
Logic Area	relative	1.00	0.80	0.63	0.51	0.39	0.32	0.25	0.20	0.16	0.12	0.10	0.08	0.06	0.05	0.04	0.03
SRAM Area	relative	1.00	0.78	0.61	0.48	0.38	0.29	0.23	0.18	0.14	0.11	0.09	0.07	0.06	0.04	0.03	0.03
50/50 Area	relative	1.00	0.79	0.62	0.49	0.38	0.30	0.24	0.19	0.15	0.12	0.09	0.07	0.06	0.05	0.04	0.03
High Performance Devices																	
Delay	ps	0.87	0.74	0.64	0.54	0.51	0.40	0.34	0.29	0.25	0.21	0.18	0.15	0.13	0.11	0.10	0.08
Average Device Capacitance	relative	1.00	0.87	0.76	0.66	0.58	0.50	0.44	0.40	0.36	0.31	0.28	0.24	0.22	0.20	0.18	0.16
Circuit speedup: 1/delay	relative	1.00	1.18	1.36	1.61	1.71	2.18	2.56	3.00	3.48	4.14	4.83	5.80	6.69	7.91	8.70	10.88
ITRS Max Clock	relative	1.00	1.30	1.78	2.11	2.38	2.90	3.39	3.86	4.42	5.45	6.42	7.63	8.75	10.22	12.00	14.05
Vdd	volts	1.10	1.10	1.10	1.00	1.00	1.00	1.00	0.90	0.90	0.90	0.80	0.80	0.70	0.70	0.70	0.70
Vdd/Vt	ratio	5.64	6.55	6.67	6.10	4.22	6.62	6.85	6.08	5.39	5.49	4.82	4.10	3.50	3.48	3.41	3.37
Power Density @ Circuit Speedup	relative	1.00	1.29	1.65	1.77	2.13	2.95	3.95	4.19	5.52	7.41	7.54	10.19	10.17	13.91	17.12	23.47
Power Density @ Max Clock	relative	1.00	1.43	2.17	2.31	2.97	3.93	5.23	5.39	7.00	9.75	10.01	13.39	13.30	17.98	23.61	30.33
Energy/Operation	relative	1.000	0.867	0.756	0.542	0.478	0.413	0.367	0.268	0.238	0.208	0.147	0.129	0.090	0.081	0.072	0.063
Low Operating Power Devices																	
Delay	ps	1.52	1.33	1.17	1.03	0.90	0.79	0.79	0.61	0.53	0.47	0.41	0.36	0.32	0.28	0.24	0.21
Circuit speedup: 1/delay	relative	0.57	0.65	0.74	0.84	0.97	1.10	1.10	1.43	1.64	1.85	2.12	2.42	2.72	3.11	3.63	4.14
Vdd	volts	0.90	0.90	0.80	0.80	0.80	0.70	0.70	0.70	0.60	0.60	0.60	0.50	0.50	0.50	0.50	0.50
Vdd/Vt	ratio	3.13	2.97	2.81	2.95	2.90	3.10	3.00	3.03	2.33	2.40	2.39	2.10	2.09	2.07	2.06	2.03
Power Density @ Circuit Speedup	relative	0.38	0.48	0.48	0.59	0.77	0.73	0.83	1.21	1.16	1.47	1.86	1.66	2.11	2.79	3.64	4.56
Energy/Operation	relative	0.669	0.580	0.400	0.347	0.306	0.202	0.180	0.162	0.106	0.093	0.083	0.051	0.046	0.041	0.037	0.032

Note: units of "relative" represent values normalized to those of the 2005 high performance technology

Figure 6.1: ITRS roadmap logic device projections

6.2 Logic Today

Perhaps the most mature technology that is in use today for logic and memory is CMOS silicon. This section discusses the outlook for this technology through the end of the next decade in two ways: in summary form as projected by the ITRS Roadmap, and then in terms of the inherent device physics as seen by industry-leading sources. The latter is covered in two pieces: silicon-related and non-silicon technologies.

6.2.1 ITRS Logic Projections

The 2006 ITRS Roadmap[13] projects the properties of silicon CMOS logic through the year 2020, and is used as a standard reference for future trends. This section overviews some of the more general projections as they may relate to Exascale logic sizings for 2015 developments. It is important to note, however, that these projections are for “business as usual” CMOS silicon, and do not represent potential alternative silicon device families.

For logic there are two kinds of devices that are most relevant: those designed for modern leading-edge high performance microprocessors, and those designed for low operating power where clock rates may be sacrificed. The differences lie primarily in the threshold voltages of the devices, and the operating conditions of typical circuits (V_{dd} and clock rate).

For reference, Figures 4.2 through 4.10 combine both historical data with their matching projections from ITRS for a variety of key parameters. Figure 6.1 then summarizes numerically some projections that are most relevant to this report. The columns represent years from 2005 through the end of the roadmap in 2020. For each year, the “feature size” (Metal 1 half-pitch) is listed.

The table itself is divided into three sub-tables: area-related, speed and power for the high performance devices, and similar speed and power for the low operating power devices. In this figure, any row marked as being in “relative” units represents the actual values derived from the roadmap, but normalized to the equivalent numbers for 2005 high performance devices. Thus the key results have been normalized to industry-standard high performance 90nm CMOS technology.

The years highlighted in green represent the values most relevant to chips that might be employed in 2015 in real systems.

6.2.1.1 Power and Energy

Throughout all the rows in Figure 6.1, power consumed (and thus dissipated) for some fixed subsystem of logic is computed using the formula (leakage power is ignored for now):

$$Power_per_subsystem = Capacitance_of_subsystem * Clock * V_{dd}^2 \quad (6.1)$$

Dividing this by the area of the subsystem yields the **power density**, or power dissipated per unit area:

$$Power_density = (Capacitance_of_subsystem/area_of_subsystem) * Clock * V_{dd}^2 \quad (6.2)$$

or

$$Power_density = Capacitance_per_unit_area * Clock * V_{dd}^2 \quad (6.3)$$

Also, canceling out the clock term in Equation 6.1 yields not the power of a circuit but (at least for pipelined function units) an **energy dissipated per machine cycle** which, if the subsystem is pipelined is the same as the **energy per operation**, and which will prove useful in later work:

$$Energy_per_operation = Capacitance_per_unit_area * V_{dd}^2 \quad (6.4)$$

It is instructive to express the capacitance of a subsystem as:

$$Capacitance_of_subsystem = Capacitance_per_device * \#_of_devices \quad (6.5)$$

which yields another variant of Equation 6.2:

$$Power_density = (Capacitance_per_device * \#_of_devices / area_of_subsystem) * Clock * V_{dd}^2 \quad (6.6)$$

However, the ratio of device count to area is exactly transistor density as discussed above, and thus we can rewrite this as:

$$Power_density = Capacitance_per_device * Transistor_density * Clock * V_{dd}^2 \quad (6.7)$$

Finally, we can also invert this last equation to yield one that indicates at what maximum clock frequency a chip could run at in order to stay at or below some power density limit:

$$Clock = Power_density / (Capacitance_per_device * Transistor_density * V_{dd}^2) \quad (6.8)$$

6.2.1.2 Area

The first three rows of Figure 6.1 reflect the effect of the technology on **transistor density** - the number of transistors per unit area on a die. In the rows, these numbers are expressed as relative “area” factors, that is by what factor would a circuit designed in 90nm technology shrink if it were simply recast in succeeding year’s technology, with no enhancements or scaling. There are separate rows for circuits that are pure logic, pure SRAM arrays, and circuits with a 50/50 mix of logic and SRAM.

The key take-away is that for 2015 deployment, using ITRS projections we can assume that a core designed initially for 90nm would shrink between a factor of 6 and 8 in area, allowing up to 6 to 8 times more of the same complexity cores on a die than in 2005.

6.2.1.3 High Performance Devices

The high performance device sub-table of Figure 6.1 represents the mainstay of modern CMOS microprocessor technology. The first row gives the **intrinsic delay** of an N-type device, with the third representing its reciprocal relative to the 2005 number. Thus a number of 3.48 in 2013, for example, implies that the same circuit using 2013 devices would be capable of running 3.48 times faster than in 2005.

The column labeled “ITRS Max Clock” represents clock rate growths projected by ITRS for a pipeline stage involving 12 invertors in a chain. This number grows somewhat faster than the second row, due to other circuit effects. The choice of a 12 inverter stage was made for historical reasons based on microprocessors designed for the maximum possible clock rate (called **super-pipelining**), regardless of microarchitectural performance. It is important to understand that such “short pipe stages” have been found since then to be inefficient when used in modern superscalar microprocessors, where long pipe lengths exacerbate significant data hazards and bottlenecks that negatively impact performance. In addition, power and clock rate are proportional, so higher clock rates also raise power dissipation. Thus, for several years the microprocessor industry has retreated from super-pipelining in favor of more efficient but lower clock rate microarchitectures. Thus this row should be taken as an indication of absolutely maximum upper end potential, not expected practice.

The row labeled “V_{dd}” represents the main operating voltage projected for use by circuits using these devices. The row below this gives the ratio between V_{dd} and the main threshold voltage of the projected devices. Over the time period of interest, this ratio is about 5.5, meaning that there is sufficient voltage for multiple devices to be stacked, and still have good operating margins.

Given these numbers, the row labeled “Power density @ Max Clock” represents the relative change in power dissipated per unit area on a die when the V_{dd} is as marked, and the clock rate is the maximum projected by ITRS. The numbers for 2013-2014 indicate that if we simply tiled dies on that time with multiple copies of a 2005 core, the die would dissipate 7-10X as much power per unit area - far more than is assumed coolable today.

The row labeled “Power Density @ Circuit Speedup” is a similar calculation but assuming circuits speed up only as much as the devices do. The growth in power dissipation, however, is still significant.

The last row in this section reflects the “energy per operation” relative to a 90nm circuit, in units of pico joules (pJ) where 1 pJ equals 10⁻¹² joules. Since this is “per operation” the clock rate in Equation 6.1 is irrelevant. Thus a circuit that took X pJ per operation in 2005 will, in 2013-2014 technology, take 1/4 to 1/5 of that.

Also we note again that numerically, if performing some operation takes *X pJ* of energy, then computing 1 “exa” (10¹⁸) of them in one second consumes *X MW* of power.

6.2.1.4 Low Operating Voltage Devices

The final section of Figure 6.1 reflects projections for devices designed to be placed into lower power circuits. The devices are designed to have a higher threshold voltage (less leakage) and run at a lower V_{dd}. Thus the circuit families cannot be as complex in terms of stacking devices, there is less margin, and the circuit is significantly slower than the high performance one, but the power density and energy per operations are significantly improved, by a factor of two in energy per operation.

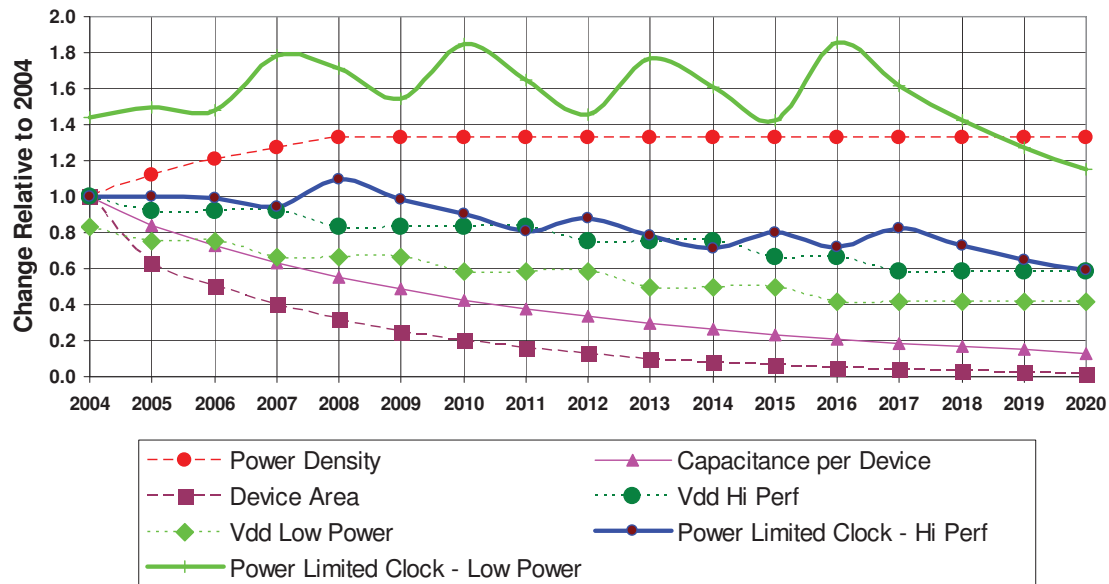


Figure 6.2: Relative change in key power parameters

6.2.1.5 Limitations of Power Density and Its Effect on Operating Frequency

Figures 4.3, 4.7, and 4.8 diagram the density, V_{dd} , and clock parameters in the power equation 6.7. These includes both historical data (from both single core chips and the newer multi-core chips), and ITRS extrapolations. Figures 4.9 and 4.10 then graph similarly observed and projected maximum power dissipated per chip, and the equivalent power density. These graphs yield the following key takeaways:

- Transistor area (density) continues to drop (increase) as the square of the feature size. This tends to increase the overall power of a chip because more transistors are present.
- V_{dd} for the dominant high performance logic families has essentially flattened after a long period of significant reductions, with minimal projected future decrease. Thus decreasing power by reducing voltage has just about run its course.
- Capacitance per device continues to decrease approximately linearly with feature size. This both allows the transistors to run faster, and reduces the power per device.
- After a multi-decade run-up, sometime in the early 2000's, the clock rate actually implemented in real chips began to lag behind that which was possible due to the inherent improvement in the transistors delay characteristics. In fact, a plateauing around 3GHz occurred at the upper end.

The reason for the last of these observations is clear from Figure 4.9 - the absolute power dissipated by a typical chip reached a threshold of around 100 watts above which it is uneconomical to cool. At this point, the only knob available to designers to limit this was operational frequency. Thus the flattening of clock rate.

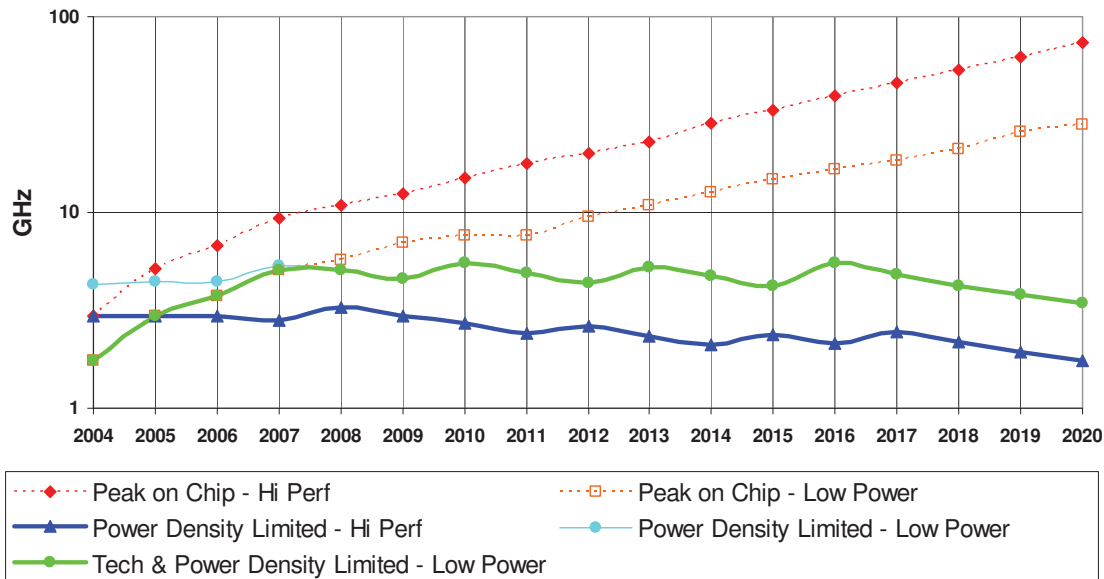


Figure 6.3: Power-constrained clock rate

An instructive exercise to explore what might happen in the future is to use these ITRS roadmap projections to forecast what kind of growth in operating frequency is thus likely to be observed. To do this we assume that chips of the future will look essentially like today in terms of mix of transistors. This is in fact more or less will happen if the current ground swell to multi-core processor chips continues unabated, with future logic chips representing a tiling of copies of what look like today's core.

Figure 6.2 does this by referencing the ITRS trends to those values of 2004, the approximate year when clock rates peaked. The curves include the maximum power density that can be cooled, the range of V_{dd} that is still available, the transistor density (the reciprocal of this is graphed to keep the numbers in the range of the others), and the capacitance per device. Then, using Equation 6.8 we can determine a clock rate that will just support the maximum possible cooling capacity. Figure 6.3 converts this relative clock curve to an absolute clock. Also included for reference is the peak clock that is possible based on the transistor delay.

These curves give a somewhat astonishing conclusion: with a design-as-usual mindset, micro-processors of the future will not only *not* run faster than today, they will actually decline in clock rate.

The same sort of analysis can be done if instead of the high performance logic we assume the low operating power form discussed in Section 6.2.1.4. Figure 6.2 includes a separate curve for the V_{dd} in this case, as does Figure 6.3.

The conclusion from this chart is equally enlightening - again the clock rates are way below the absolute possible based on the transistor performance. However, because of the lowered voltage, very shortly the maximum clock that is sustainable at the maximum power dissipation actually becomes *higher* than that for the supposed high performance logic. This may be an indication that the low voltage logic is perhaps a better choice for Exascale than conventional logic.

High Volume Manufacturing	2008	2010	2012	2014	2016	2018	2020	2022
Technology Node (nm)	45	32	22	16	11	8	6	4
Integration Capacity (BT)	8	16	32	64	128	256	512	1024
Delay Scaling	>0.7 ~1?							
Energy Scaling	~0.5 >0.5							
Transistors	Planar 3G, FinFET							
Variability	High Extreme							
ILD	~3 towards 2							
RC Delay	1	1	1	1	1	1	1	1
Metal Layers	8-9	0.5 to 1 Layer per generation						

Figure 6.4: Technology outlook

6.2.2 Silicon Logic Technology

The ITRS data of the prior section focused on general trends; given the challenges involved, it is instructive to understand the physics behind these trends, and how they affect the kinds of circuits and performance metrics that make up the trends. The following subsections discuss the underlying scaling challenges, the potential for new processes such as SOI, and what this means to logic and associated high speed memory circuits.

6.2.2.1 Technology Scaling Challenges

We begin with the technology outlook presented in Figure 6.4. Transistor integration capacity is expected to double each generation and there are good reasons to believe that it will be on track. Logic and circuit delay scaling, however, has already slowed down, and is expected to slow down even further, approaching a constant. Energy scaling too has slowed down, so transistor architecture will have to change to something other than today's planar architecture, and thus the variability in transistors will become even worse than what it is today. In a nutshell, you will get transistor integration capacity in the future (the first major benefit), but not the same performance, and the energy/power reduction.

Figure 6.5 is a simplified transport model describing the scaling challenges. Thinner gate dielectric (gate oxide) is better since it results in higher gate capacitance, creating higher charge volume. But gate oxide scaling has reached a limit due to tunneling, causing excessive gate leakage. High-K gate dielectric is a solution, but for just a generation or two, since it too has to scale down and will reach the scaling limit.

Lower threshold voltage (V_t) is desired for higher current, but it causes excessive source to drain sub-threshold leakage, that is why, we suspect that V_t scaling too has reached the limit. Mobility engineering, such as straining, to improve the drive current will continue, but with diminishing return.

To summarize, due to gate dielectric scaling and V_t scaling slowing down, the supply voltage

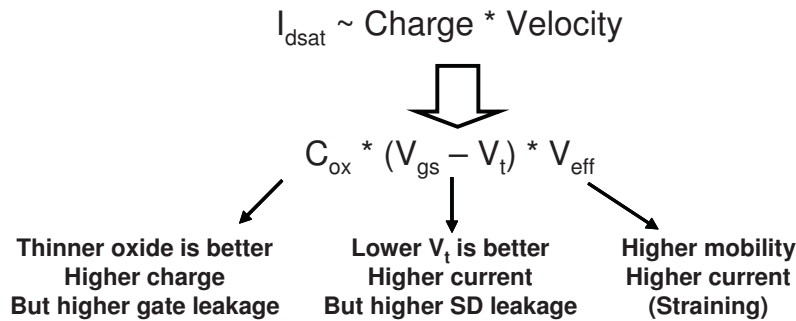


Figure 6.5: Simple transport model

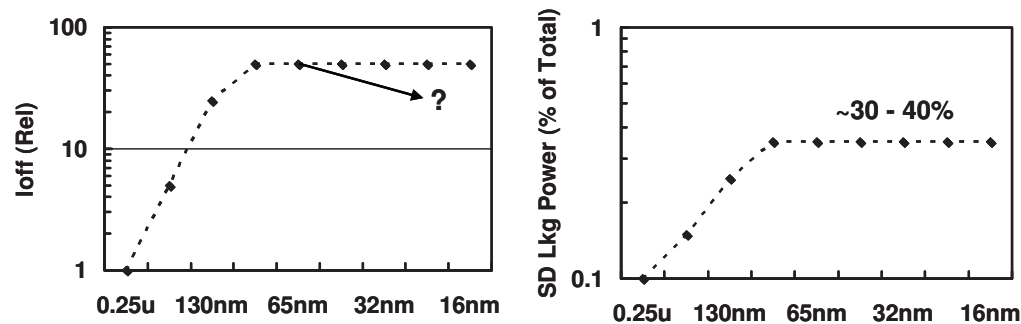


Figure 6.6: Transistor sub-threshold leakage current and leakage power in recent microprocessors

scaling too will slow down, transistor performance increase will slow down, and active energy/power reduction will slow down.

Figure 6.6 graphs increase in sub-threshold leakage current in successive past technology generations, and corresponding increases in the leakage power as a percentage of total power. The slope of the non-flat part of the I_{off} curve is often called the **sub-threshold slope**, and the flatter the curve, the less that future generations of technology will have to worry about leakage. Also included are extensions to the future. Notice that the transistor leakage power increased exponentially to deliver the necessary performance, but now it is staying constant, and may even decrease to keep full chip leakage power under control.

Figure 6.7 shows predictions of technology and attributes for the timeframe of Exascale systems as envisioned here.

In terms of summary projections for future designs in silicon such as in Section 7.3, it is thus reasonable to assume that leakage power will account for 30% of the total chip power.

6.2.2.2 Silicon on Insulator

There is a common belief in the technical community that something drastic can be done in the process technology to reduce power further, and the most commonly cited solution is **SOI (Silicon on Insulator)** technology, where the transistor is built not on a conductive substrate (as in **bulk silicon**), but as a film on an insulating oxide. This technology comes in two flavors: **partially depleted**, and **fully depleted**. In partially depleted SOI, the silicon film is thick and only part

Tech Node	Units	32nm	22nm	16nm	Comments
High Volume		2010	2012	2014	
Vdd	Volts	1	0.95	0.9	Vdd scaling slowed down
Delay Scaling		1	-15%	-10%	Delay scaling slowed down (estimated)
FO4 Delay	ps	10	8.5	7.65	Estimate
Logic density	MT/mm2	1	1.5	2.2	1.5X, limited by design rule complexity and interconnects
Cache (SRAM) density	MB/mm2	0.2	0.34	0.54	Includes tags, ECC, etc 1.6X limited by stability
Logic Cdyn ($\alpha C/\text{Tran}$)	nf/MT	0.2	0.16	0.13	0.8X scaling due to variability, etc.
Cache Cdyn ($\alpha C/\text{MB}$)	nf/MB	0.09	0.06	0.04	0.8X scaling

Figure 6.7: Technology outlook and estimates

of the body of the transistor is depleted (empty of free carriers) during inversion, as opposed to fully depleted SOI where the silicon film is very thin and the region below the whole body of the transistor is depleted of free carriers.

There are two major reasons for the claims about power savings in SOI, namely reduced source-drain junction capacitance, and better sub-threshold slope. We will examine each of these separately.

It is true that the source-drain junction capacitance is lower in SOI due to shallow junctions, which do not extend all the way into the substrate as in the case of bulk transistors. But, bulk transistors, on the other hand, use compensating implants to reduce this capacitance. Moreover, source-drain junction capacitance contributes relatively little to the overall power consumption. Therefore, the benefit of lower junction capacitance in SOI has limited or no impact on power in comparison to modern logic CMOS.

Better sub-threshold slope does permit reducing the threshold voltage of a transistor, providing higher performance for the same leakage power, or lower leakage power for the same transistor performance. Partially depleted SOI and bulk transistors both have comparable sub-threshold slope, and thus comparable performance and leakage power. Fully depleted SOI, on the other hand, shows improved short-channel effects, and much better sub-threshold slope. Research and development activities are already underway in the industry to exploit this. For example, **FINFETs** or **Tri-gate transistors** are inherently fully depleted, and will exhibit this benefit. The power and energy benefit is estimated to be in the 10% to 20% range, and does not constitute orders of magnitude improvement desired for Exascale.

6.2.2.3 Supply Voltage Scaling

This section considers the use of **supply voltage scaling** to reduce power and energy, with benefits as well as design challenges. This is perhaps the biggest available lever that is currently available in conventional silicon technology.

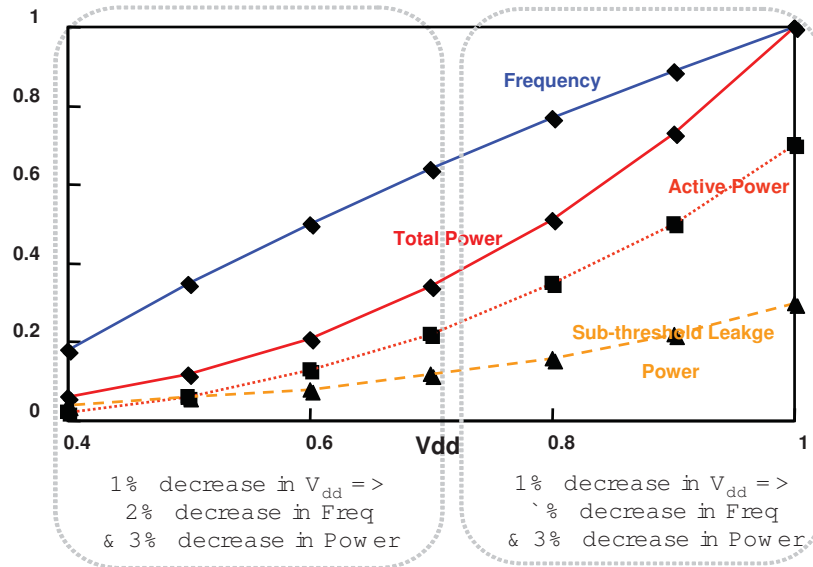


Figure 6.8: Frequency and power scaling with supply voltage

As background consider some circuit that is designed to run at some maximum clock rate when powered by some nominal V_{dd} . As discussed in Section 6.2.1.1, the circuit power is proportional to the square of this voltage, so decreasing it has a significant effect. However, as V_{dd} is decreased, neither the RC characteristics nor the threshold voltage of the transistors making up the circuit change much, meaning that it takes longer for a signal (say from a 0 to a 1) to reach the threshold of the transistors that it is driving. Thus to maintain correct operation of the circuit, the clock must be decreased appropriately. While this decreases the performance of the circuit, it also approximately proportionately decreases the power even further. However, the reduction in performance is lower than the savings in power and energy.

This benefit is shown quantitatively in Figure 6.8, based on an analytical model. As supply voltage is reduced, frequency reduces, but so do the active and leakage power. At higher supply voltage (much above the threshold voltage of a transistor), reduction in frequency is almost linear with voltage. Thus a 1% drop in V_{dd} requires a 1% drop in clock (and thus performance), but saves 3% in active power (the power reduction is cubic). As the supply voltage is lowered even further (close to threshold voltage of the transistor), peak frequency must decrease somewhat faster. Power savings, however, is still higher than frequency loss. Here a 1% drop in V_{dd} requires a 2% drop in clock, but still yields a 3% drop in active power.

Figure 6.9 shows measured power and energy savings in an experimental logic test chip in a 65nm process technology. Nominal supply voltage is 1.2V, and as V_{dd} is scaled down, operating frequency and total power consumption reduce as shown in Figure 6.9(a). Figure 6.9(b) shows energy efficiency as you reduce the supply voltage, where **energy efficiency** here is defined as the number of clock cycles (in billions) of useful work per watt (GOPS/Watt). Notice that the energy efficiency continues to increase with reduction in supply voltage, peaks just above the threshold voltage of the transistor (at about 320 mV in this case), and then starts declining. When the supply voltage nears the threshold voltage, reduction in operating frequency is more severe, compared to reduction in power, and therefore energy efficiency drops.

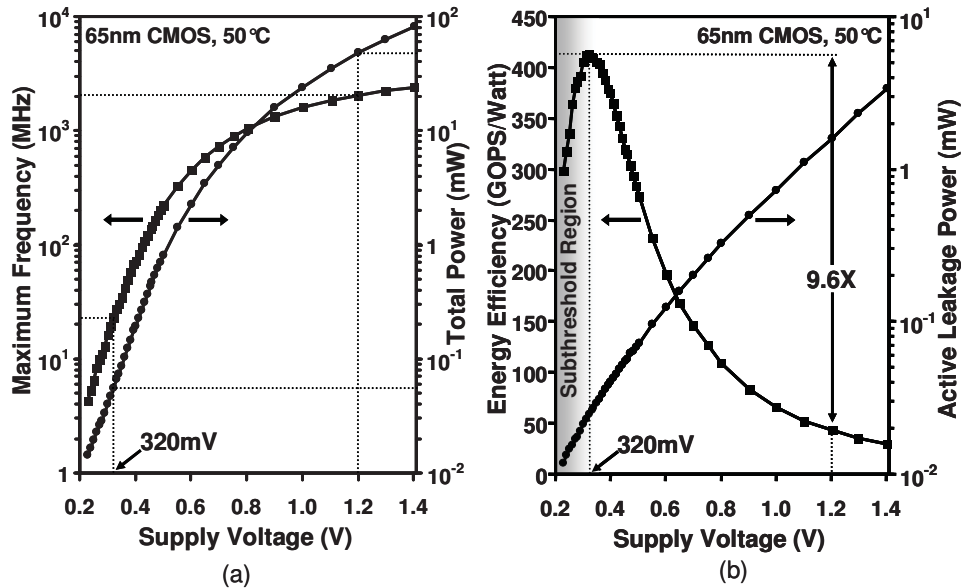


Figure 6.9: Sensitivities to changing V_{dd} .

The downside to this increase in efficiency, at least for Exascale, is that if performance approximating that achieved by the circuit at nominal V_{dd} is desired by reduced voltage implementations, then a significant increase in circuit parallelism is needed. For example, in Figure 6.9, we see that there is approximately a 100X reduction in clock from 1.2V to 320 mV. This would require at least 100 copies of the same circuit to achieve the same performance, with a 100X area cost, and even though the overall savings in power is about 10X.

Figure 6.10 shows how supply voltage scaling benefit will continue even with technology scaling. It plots simulated energy efficiency for the above logic test chip in 65nm, 45nm, and 32nm technologies, with variation in threshold voltage (V_t) of 0 and ± 50 mV. Notice that the energy efficiency continues to improve with supply voltage scaling, peaks around the threshold voltage, and has weak dependence on threshold voltage and variations.

In summary, supply voltage scaling has potential to reduce power by more than two orders of magnitude, and increase energy efficiency by an order of magnitude, but at some significant area penalty.

6.2.2.4 Interaction with Key Circuits

Although aggressive supply voltage scaling benefits energy efficiency and power, it also warrants different design practices. That is why most past and present designs do not support supply voltage scaling beyond about 30% lower than nominal. The biggest culprits in terms of circuit types that are difficult are:

- small signal arrays such as memory, caches, register files,
- dynamic logic such as Domino circuits,
- and large fan-in static gates.

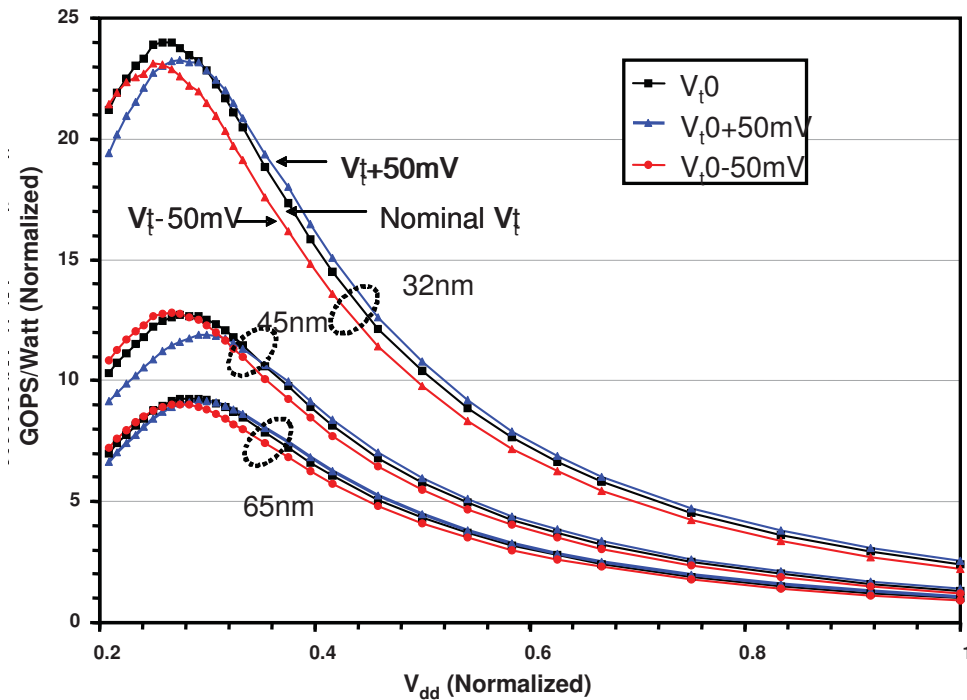


Figure 6.10: Technology scaling, V_t variations, and energy efficiency.

Static RAM (SRAM) circuits make up the backbone of a logic chip's need for information storage at almost all but the main memory level. SRAM cells today become unstable at lower supply voltage because they are designed with small transistors for higher performance and lower area. These cells can be designed to work at lower supply voltages by increasing transistor sizes and for full supply voltage (rail to rail) operation. This sacrifices some performance and area, but at a lower voltage, the logic operates at lower frequency anyway, hence performance loss is not a major issue, but the area penalty is, which could be around 10% or so.

Similarly, **Register Files** can be redesigned to operate over a wide range of voltages by replacing Jam-Latches by clocked latches. Again the cost is area and performance.

Domino logic is employed in many of today's chips to increase frequency of operation at the expense of very high power, and thus should be avoided in future designs. Large fan-in gates tend to lose disproportionate amount of performance at lower supply voltages due to transistor body effect. Therefore, designs should avoid using large fan-in gates in the critical paths of the circuits. Overall, designing for low supply voltage operation will be different from what we do today, but not difficult.

6.2.3 Hybrid Logic

Although there are several interesting proposals for performing logic operations using quantum cellular automata (QCA), DNA, diverse types of molecules, optical components, various quantum systems implementing qubits, and other exotic physical processes, none of these have a possibility for contributing to an enterprise-scale computing system within the next ten years. However, there are a few hybrid technologies that involve integrating scaled silicon CMOS with nano-scale

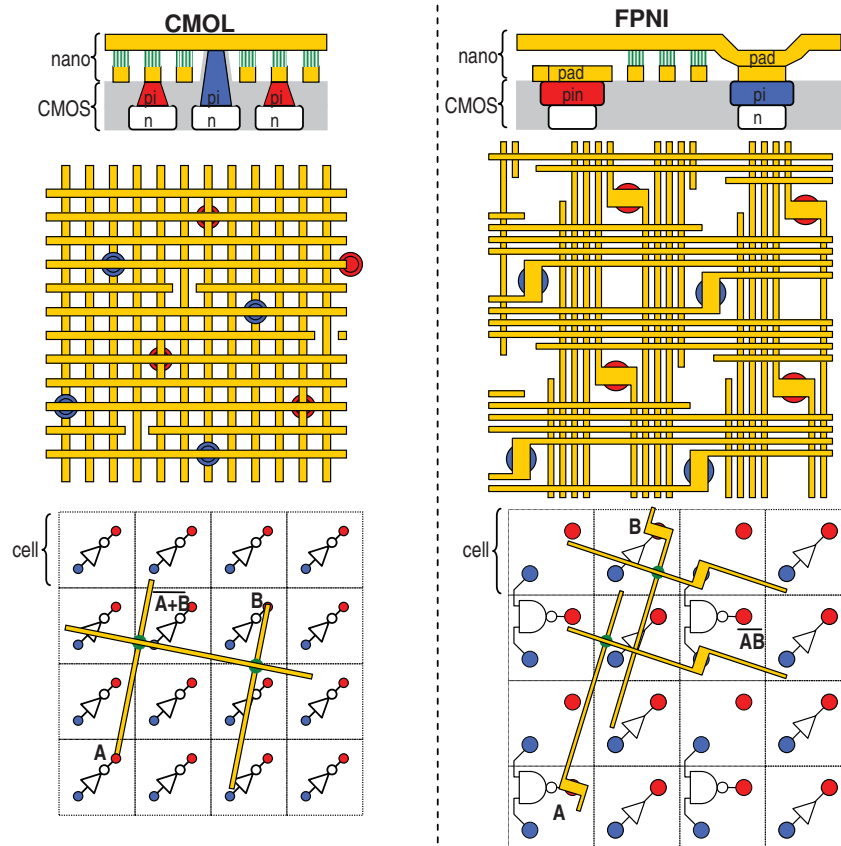
Circuit	Area tm ²				Critical Path Delay ns			Dynamic Power mW	
	CMOS	CMOL 9 nm	FPNI 30 nm	FPNI 9 nm	CMOS	FPNI 30 nm	FPNI 9 nm	FPNI 30 nm	FPNI 9 nm
alu4	137700	1004	17513	5026	5.1	6.53	28.7	0.48	0.061
apex2	166050	914	18983	5448	6	7.10	32.5	0.47	0.059
apex4	414619	672	13457	3862	5.5	5.98	27.1	0.44	0.054
clma	623194	9308	78020	22391	13.1	19.70	85.5	0.78	0.103
diffeq	100238	1194	18983	5448	6	6.86	30.6	0.33	0.044
elliptic	213638	4581	43493	12482	8.6	12.48	56.1	0.50	0.066
ex1010	391331	3486	41252	11839	9	10.03	44.4	0.84	0.106
ex5p	100238	829	11050	3171	5.1	5.42	23.8	0.37	0.047
frisc	230850	4199	43493	12482	11.3	14.02	61.8	0.52	0.068
misex3	124538	1004	14750	4233	5.3	5.52	25.7	0.50	0.061
pdc	369056	4979	48153	13819	9.6	12.74	58.0	0.90	0.110
s298	166050	829	20513	5887	10.7	12.74	58.5	0.25	0.032
s38417	462713	9308	84220	24170	7.3	12.94	63.1	0.93	0.114
s38584.1	438413	9872	66329	19036	4.8	7.80	39.4	1.29	0.153
seq	151369	1296	17513	5448	5.4	6.55	28.9	0.51	0.066
spla	326025	2994	43493	12482	7.3	10.92	48.6	0.84	0.108
tseng	78469	1194	17513	5026	6.3	7.10	29.0	0.25	0.037
total	4494491	57663	598728	172250	126.4	164.45	741.6	10.18	1.29
relative	1.0	0.013	0.133	0.038	1.0	1.30	5.87	1.0	0.13

Data under CMOS and CMOL columns from [142] and FPNI columns from [137]

Table 6.1: Some performance comparisons with silicon.

switching components presently under investigation that could provide higher effective logic density, enhanced performance per power input and/or improved resilience in the time frame of interest.

The first such hybrid technology is to use nonvolatile switches in a nano-scale crossbar array to act as logic devices or configuration bits and switches for a field-programmable gate array (FPGA), as pictured in Figure 6.11. Likharev and Strukov[94] originally proposed this type of architecture, which they named “**CMOL**” to denote a hybrid integrated system that combines molecular switches at the junctions of a crossbar to implement wired-OR functions fabricated on top of a fairly standard CMOS circuit that contains inverters. These researchers executed a series of rigorous simulations of their architecture for a set of 20 benchmark algorithms, and observed performance enhancements of approximately two orders of magnitude in area required to implement an algorithm compared to a standard FPGA architecture that used the same level of CMOS. However, given the aggressive nature of their assumptions (e.g. 4.5 nm wide nanowires in the crossbar), their approach appears to be more than 10 years out. By relaxing some of the assumptions (e.g. using 15 nm wires in the crossbar array, which have already been demonstrated experimentally), designing a more manufacturable process for connecting the nanowires to the CMOS plane, and using the nonvolatile nanoswitch junctions only for configuration bits and connections, Snider and Williams[137] provided a detailed design study for a related architecture they dubbed **field-programmable nanowire interconnect (FPNI)**. In computer simulations of the same set of benchmarks for their structure, they observed an order of magnitude increase in performance (e.g. area of a chip required for a computation) over a CMOS-only FPGA even in the presence of up to 50% defective switches in the



Schematic diagrams of hybrid logic circuits. The CMOL design by Likharev and Strukov (left column) places a nanowire crossbar on top of a sea of CMOS inverters. The crossbar is slightly rotated so that each nanowire is electrically connected to one pin extending up from the CMOS layer. Electrically-configured, nonlinear antifuses (green, bottom panel) allow wired-OR logic to be implemented, with CMOS supplying gain and inversion. This is a very high-density design that would not likely be implementable before 2020. FPNI (right column) places a sparser crossbar on top of CMOS gates and buffers. Nanowires are also rotated so that each one connects to only one pin, but configured junctions (green, bottom panel) are used only for programmable interconnect, with all logic done in CMOS. This version was designed to be implementable with today's technology, and is currently in development for space-based applications.

Figure 6.11: Hybrid logic circuits

crossbar, thus realizing both a significant performance improvement and resiliency in a technology that is feasible by 2017. Table 6.1 summarizes these projections.

Other studies of this class of nano circuits include [139], [89], [134], [122], [36], [37], [154], [91], [96], [148], [121], [64], [67], [68], [116], [135], [133], [136], [101], and [49].

A second class of hybrid structures that have been studied are **Programmable Logic Arrays (PLA)**[52][53][35][153][164][34]. These structure are basically AND/OR arrays that can implement any n-input m-output Boolean function by “programming” various points in the arrays.

Although these studies relate specifically to FPGA-type architectures, there are potential applications of this technology to Exascale computing. At the present time, there is an effort in place to build a “FPNI” chip for use primarily in space applications where size, flexibility and resilience are at a premium (especially radiation-damage tolerance). By employing different engineering trade-offs, the issues of power and speed could be optimized. A multi-core chip could have some FPGA-like cores to perform specialized functions for the chip, much as many high performance machines today have FPGAs as components. Another possibility is that the main processors cores could incorporate some favorable aspects of the FPNI technology to create hybrid architectures that are more efficient and resilient than today’s processors. Adapting such an approach to build specialty cores or introduce some of this technology into “regular” processor cores is a research opportunity for Exascale systems.

6.2.4 Superconducting Logic

Perhaps the most extensively studied non-silicon logic technology uses extremely fast magnetic flux interactions within super-cooled (around 4°K) superconducting **Josephson Junction (JJ)** devices. This technology, in the form of **Rapid Single Flux Quantum (RSFQ)** devices, was the starting point for the HTMT[47] petaflops system project in the late 1990s, and has seen a series of prototype developments since then.

Most recently, a major report written in 2005[3] provided a summary and potential roadmap for the technology. The primary technology conclusion was that if an investment of between \$372M and \$437M had been made, then by 2010 the technology would have matured to the point where a peta scale design could be initiated. This investment would have targeted a cell library and CAD tool set for RSFQ circuits, a single MCM 1 million gate equivalent processor running at 50 GHz and including 128KB of RAM, and a viable fab facility. The major technical issues that were surfaced in the report included (each discussed in detail below):

- providing memory that is dense and fast enough to support the enhanced logic speeds of the devices,
- developing architectures that were very latency tolerant,
- and providing very high bandwidth communications into and out of the cryogenic cooler.

Although the logic speeds are impressive, as discussed below the density, system level power, and difficulty in transferring data in and out of the required cryostat are all limiters to the technology’s general usefulness in across-the-board Exascale systems, except in specialized applications.

The report also draws a possible roadmap for this logic if the investment had been made in 2006, summarized in Table 6.2. As a reference point, a full 64 bit floating point unit in CMOS may take upwards of 50K gates, without much in the way of supporting register files or control logic. Thus the technology marked as “2010” might support 20 such FPUs per 1 cm² die, for about 1000

Time Frame	Device Density	Clock Rate	nanoWatts/GHz/Gate
2005	600K JJs/cm ²	20GHz	16
2010	1M JJs/cm ²	50 GHz	8
post 2010 90 nm	250M JJs/cm ²	250GHz	0.4

Table 6.2: 2005 projection of potential RSFQ logic roadmap.

GFlops potential. The “90 nm” technology, if achieved, might pack 500 FPU’s for a potential of 125 Tflops per cm².

Density and its scaling into future feature sizes is discussed in [23]. The major factor controlling density in JJs is the current densities in the superconducting wires that generate the magnetic fields needed for the junction switches – doubling the current density allows matched stripline widths to narrow by the square root of 2. These wires must now carry current densities of significant magnitude, which limits how small their cross sections can get, and are subject to the Meissner effect for proper operation. Together with the lack of multiple levels of interconnect such as found in silicon, this limits the ultimate sizes to which such devices can be reduced.

6.2.4.1 Logic Power and Density Comparison

The report also summarized several test devices that had been fabricated as of that time. One was an 8 bit serial microprocessor CORE-1 prototype[102][62] demonstrated at 21 GHz local and 1 GHz system, that dissipated about 2.3 mW in the cryocooler. This is equivalent to between 109 and 2300 nanoWatts per MHz, depending on the actual basis of overall performance.

A second demonstration was of an 8 bit parallel microprocessor FLUX-1[24] designed to run at 20 GHz, with a power of 9.2mW, and utilizing 63,107 Josephson junctions on a 10.3 mm by 10.6 mm chip in 1 1.75 μ m junction feature size. This design corresponded to a power of about 460 nanoWatts per MHz.

As a point of comparison, a modern and complete 32 bit core in a 90nm technology¹ consumes about 40 μ W per MHz, in an area of as little as 0.12mm² when implemented as a synthesized, not custom, design. This includes a multiplier array. In a 2010 technology this might translate into an area of about 0.03mm² and a power of about 12 μ W per MHz. In a 2014 28 nm technology, the design might translate into an area of about 12 μ m² and a power of about 3 μ W per MHz.

In terms of area, the report predicted that with the proposed effort a 2010 density of perhaps 1 million JJs per mm² would be achievable, which would reduce the FLUX-1’s 63K JJs to about 6.3mm². Throwing in a factor of 4 for the 8 bit to 32 bit difference, and another factor of 2 for the multiplier array, this implies that 2010 silicon might be functionally around 200 times denser, with 2014 era silicon raising this to over 500 times denser. Even assuming that the projected ultimate potential of 250M JJs per cm² was achievable, 2014 silicon would still hold a very significant density lead, and that would improve by another significant factor before silicon runs out.

In terms of power, if we again inflate the power of the FLUX-1 by a factor to allow for a more fair comparison to a full 32 bit core, then we get a number of about 3.7 μ W per MHz - about the same as that for silicon in 2014. Using the 2X reduction listed in Table 6.2 from [3], and adding another 2X to approximate the jump from the FLUX-1 technology to the 2005 technology, gives perhaps a 3-4X advantage to the RSFQ.

¹Based on a MIPS core as described at <http://www.mips.com/products/cores/32-bit-cores/mips32-m4k>

6.2.4.1.1 Cooling Costs The above comparison is for the logic alone, not the communication with memory, and perhaps more important, the losses in the cooler needed to keep a RSFQ circuit at 4°K. The second law of thermodynamics specifies that the minimum power required (Carnot) to absorb a watt of heat at 4°K if everything is perfect is given by $(T_h - T_c)/T_c$ where T_h is the hot (ambient) temperature and T_c is the cold temperature. For 300°K and 4°K this Carnot specific power becomes 74 W/W, meaning that 74 W must be expended to keep a 1 W source at 4°K.

In real life, the current state of the art is much worse than this, with typical numbers in the range of 6.5 kW to cool 1.5 W.² If this scales to the larger powers one might need for a full-blown Exascale system, it represents a 4300 to 1 multiplier over the logic power.

6.2.4.2 The Memory Challenge

Architecturally, the first major challenge is getting enough memory close enough to such logic to support such computational rates. As of 2005, densities of RSFQ RAMs had been demonstrated at about 16kb per cm². As reference, DRAM of that generation ran about 1.4 gigabits per cm² - about 1 million times denser. CMOS SRAM, at about 1/40 of DRAM density, was still 25,000X denser. The roadmap called for a 1 Mbit per cm² RSFQ memory chip in the 2010 timeframe - about 6000X less dense than DRAM at the time, and 125X less dense than SRAM. Alternative memory technologies, especially MRAM, were also proposed to help out, but required placing them in a “warmer” (40-70°K vs 4°K for the logic). In either case, this still requires a huge amount of silicon (with non-trivial power demands) to be placed in a cryostat if memory densities comparable to those discussed in Sections 7.2.1 or 7.3 are needed.

6.2.4.3 The Latency Challenge

A very related challenge is the latency within such systems. RSFQ is a logic that is inherently pipelined at virtually the device level, meaning that even functional pipelines can grow to the hundreds of cycles, and off chip references even further. Keeping a pipeline with hundreds of stages requires hundreds of independent sets of data to initiate through them, which in turn requires architectures that are much more vector-oriented than current generations of microprocessors. Further, at 250 GHz, a memory chip that is say 100 ns away in the warmer part of the cryo would be 25,000 cycles away from the core. For a typical “byte per flop” of memory data, this would mean that the processing cores might have to generate and track on the order of up to 3,000 concurrent memory references per FPU; even the relaxed taper of the strawman of Section 7.3 still translates into perhaps 500 concurrent memory references. Both of these are far in excess of current practice, and would require very significant amounts of buffering and comparison logic to track them.

6.2.4.4 The Cross-Cryo Bandwidth Challenge

The final challenge is in getting enough bandwidth in and out of each cryostat. This is particularly relevant for systems that may want to grow to multiple cryo systems, such as in a data center scale Exasystem. The problem is in leaving the very cold areas of the cryostat. Doing so with metal wires represents a huge cooling problem; thus optical techniques were proposed in the report. This, however, has the problem of going directly from RSFQ to optical at 4°K. Potential solutions might involve wire from the 4°K to the 70°K region where the memory might live. As is discussed elsewhere in this report (such as in Section 7.5), however, there are no known interconnect technologies that would not require massive amounts of power to be dissipated within the cryo.

²See for example the Janis CSW-71D, <http://www.janis.com/p-a4k14.html>

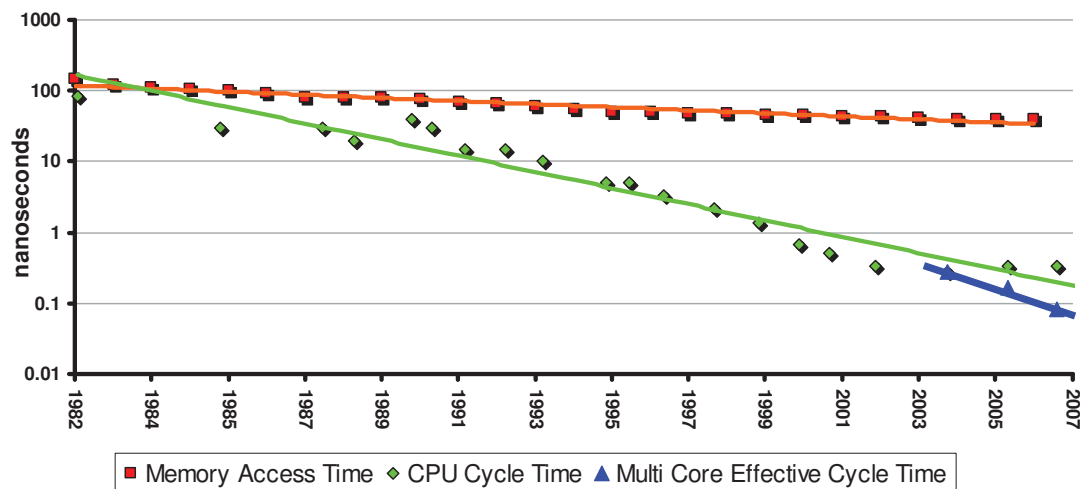


Figure 6.12: CPU and memory cycle time trends.

6.3 Main Memory Today

This section discusses technologies appropriate for use for main memory in computing systems. While the main focus is on today's SRAM, DRAM, and various forms of flash, also covered are some emerging technologies where there is enough of basis to project potential significant commercial offerings in a reasonable time frame.

6.3.1 The Memory/Storage Hierarchy

Today's computers are characterized by having central processors, connected to a hierarchy of different memory types. It has been an evolutionary process getting the semiconductor component industry to this place. Indeed, if one looks at the architecture of the earliest computers the storage hierarchy was essentially flat, yet the hints of hierarchy were there. The Hollerith card decks sitting in file racks of the machine room might have been the first mass storage technology.

Today the situation is more complex. This increased complexity is the result of technological evolution bounded by economics. The technological evolution is brought about by the steady progression of semiconductor scaling and is often expressed in terms of Moore's Law, which states that the density of transistors on a chip doubles every 18 months or so, and these transistors are inherently faster. Economics govern the application of this density increase. The economics of, say, a CPU vendor has in the past often driven that vendor to apply those transistors towards increased performance per cycle, and an increasing clock rate. On the other hand, the economics of a memory vendor drives that vendor to increase the density of the memory devices produced, without significant increases in performance. Similarly, economics govern the evolution of rotating magnetic storage so that once again it is density that is increased, with little improvement in performance.

The result of this evolution is that gaps in performance grow between the CPU and memory, and between memory and rotating storage. Figure 6.12 shows how CPU cycle times have diverged from main memory cycle times over the past 25 years. This difference is now approaching a 1000X, and is colloquially called the "memory wall."

Cell Size (μ^2)	Tech Node (nm)	Cell Size(F^2)
IBM/Infineon MRAM		
1.42	180	44
Freescale 6T-SRAM		
1.15	90	142
0.69	65	163
Intel 65nm process 6T-SRAM		
0.57	65	135
Freescale eDRAM		
0.12	65	28
Freescale TFS: Nanocrystalline		
0.13	90	16
Micron 30-series DRAM		
0.054	95	6
Samsung 512Mbit PRAM Device		
0.05	95	5.5
Micron 50-series NAND		
0.013	53	4.5

Table 6.3: Area comparisons of various memory technologies.

In order to maintain the performance of the CPU in the face of this growing gap, the processor vendors have evolved increasingly complex level 1 and level 2 caches, using large portions of their Moore's Law provided transistors to build larger SRAM caches.

6.3.2 Memory Types

In evaluating the suitability of a memory technology for a given application it is helpful to have several metrics by which said memories can be compared. For the purposes of this section, the metrics we will look at are;

1. The speed of the memory;
2. The silicon area required to make the memory;
3. The fault-tolerance of the memory;
4. The power characteristics of the memory.

Because area has been the predominant metric, Table 6.3 lists for relevant technologies and their relative areas per bit, where the term "F" is the semiconductor feature size. Figure 6.13 then graphs the equivalent numbers as projected by the ITRS roadmap on a relative basis, where 1.0 is the relative density of DRAM at that time.

6.3.2.1 SRAM Attributes

Static RAM (SRAM) cells are typically constructed from 6 transistors and are most commonly used for today's fastest memory circuits. These cells make up the L1 and L2 cache memories of today's CPUs and can account for over 65% of the die area of some CPUs. The 6 transistors

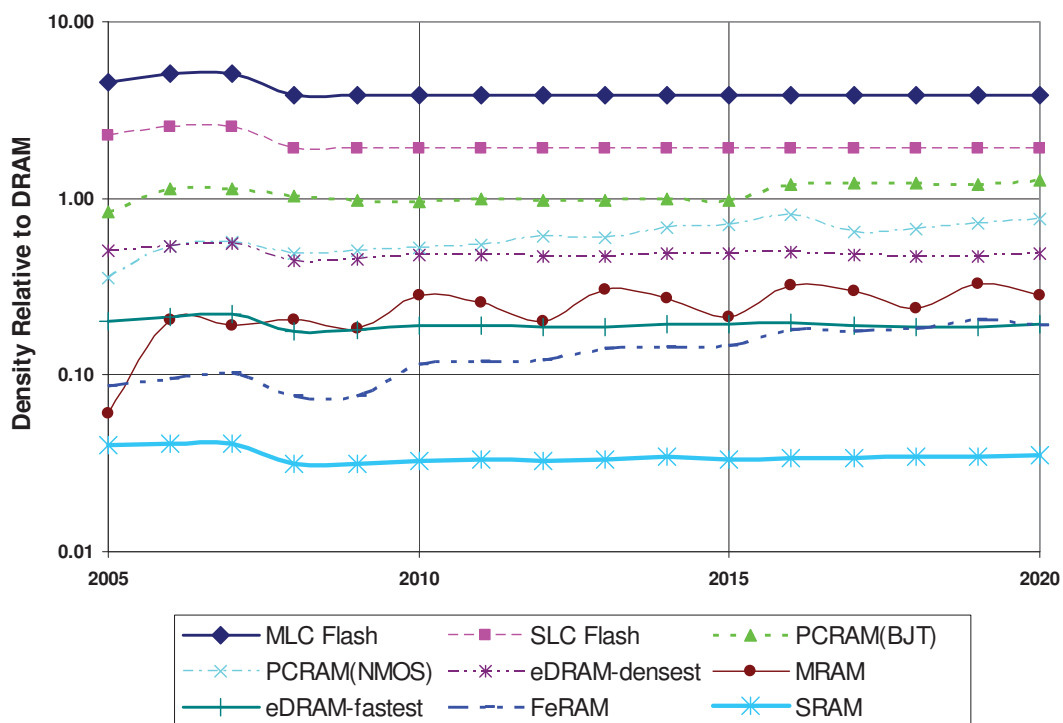


Figure 6.13: ITRS roadmap memory density projections.

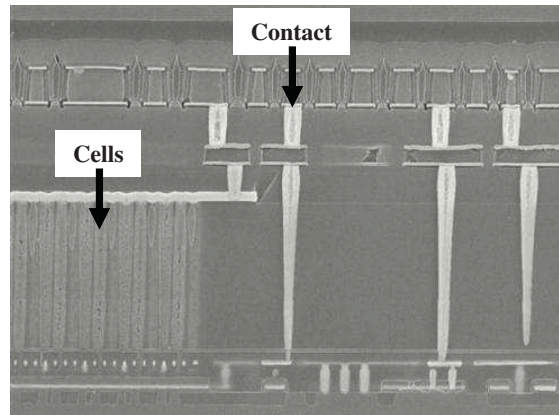


Figure 6.14: DRAM cross section.

and their associated routing makes the SRAM cell one of the largest memory cells. Most SRAM cells are in the range of $140\text{-}150\text{F}^2$. The SRAM cell is a bi-stable latch and requires power to be maintained in order for the cell contents to remain valid. In addition, SRAM cells are subject to radiation-induced failures that affect their **soft error rate (SER)**, and must be carefully designed with additional ECC bits and a layout that ensures that an SER event does not affect multiple bits in the same data word. SRAM cells may be designed for low power or for high performance. The memories used in CPU caches obviously take the later approach and are thus substantial consumers of power. Approaches such as segmenting the power and reducing voltages to the portions of the array not being addressed to help mitigate SRAM power consumption.

6.3.2.2 DRAM Attributes and Operation

DRAM cells use a capacitor as a storage element and a transistor as an isolation device. In a read operation, the transistor allows the charge on a cell to be placed onto the bit-line, which is sensed by the sense amp and converted to a one or zero. The sense amplifier also boosts the bit-line to a higher voltage, which thus restores the charge on the cell. This read operation is therefore a destructive operation. In a write operation the cell is either drained of its charge or supplied with a charge through the access device and bit-line[79].

There are two types of DRAM cell structures used in commodity DRAM devices. The **stacked cell DRAM** uses a capacitor built above the silicon. The **trench cell DRAM** uses a capacitor built into the silicon. Each has its advantages and its disadvantages, but both face some similar challenges in the next few years. The size of these DRAM capacitors is in the range of $20\text{-}30$ femto-Farads and it is the ratio of this capacitance to the capacitance of the bit-line that determines the reliability with which these cells can be read. To maintain a reasonable ratio that can be sensed rapidly and accurately the value of this cell capacitance must be maintained. As DRAM cells are typically 6F^2 or 8F^2 , there is very little room to build the cell. This is requiring cell aspect ratio to increase at an accelerating rate. The SEM image in Figure 6.14 shows a typical stacked memory cell and the high aspect ratio of that cell.

As the aspect ratio increases it becomes more difficult to etch and fill the trench capacitors. Stacked cell capacitors suffer from toppling and fill difficulties as the aspect ratio increases. Advances in dielectrics and reductions in bit-line capacitances are needed to continue advanced DRAM

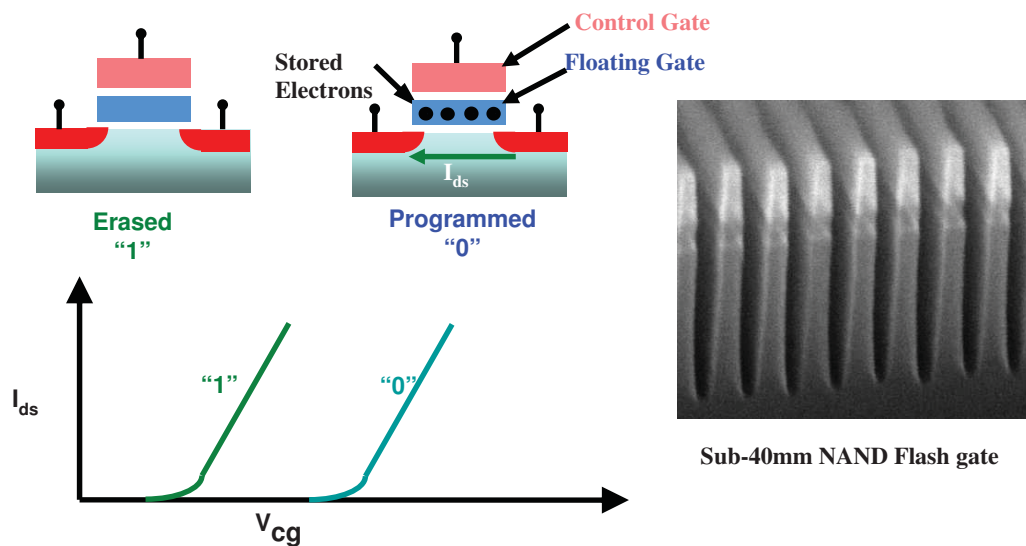


Figure 6.15: Programmed and un-programmed NAND cells.

scaling.

As shown in Figure 6.12, commodity DRAM access times have not kept pace with processors. This is not to say that fast DRAM cannot be built. **Embedded DRAM**, **Fast Cycle DRAM** and **Reduced Latency DRAM** all demonstrate fast DRAM devices. However, all of these carry a substantial die size penalty. In order to achieve the performance, the arrays become less efficient. Each of these fast DRAMs has enjoyed some success, however, although all three have been available for years not one computer maker has elected to use one of them as a main memory technology. Each, however, has merits when used as a cache memory.

6.3.2.3 NAND Attributes and Operation

NAND Flash memory is a non-volatile memory and operates by trapping electrons on a secondary gate structure of a MOS transistor (see Figure 6.15). This secondary gate is a floating gate and receives its charge when the gate voltage is elevated enough for electrons to tunnel onto the floating gate. Here, the charge is trapped and biases the junction when the cell is read.

The gate dielectric and structure have been carefully engineered to retain the trapped electrons for 10 or more years. Still, NAND systems must be carefully designed so that writing nearby bits do not affect previously written data and such that reading the cells does not allow trapped charge to leak from the floating gate[92].

NAND devices have been highly optimized for mass storage applications and are block oriented devices. A typical 16Gb NAND device will be organized as 4096 4Mbit blocks. Individual cells cannot be erased or written, but rather, entire blocks must be erased at a time.

In addition to its non-volatility, NAND has the advantage of having small cells. Typical NAND cells are $4F^2$ and with today's **Multi-Level-Cell (MLC)** technology these cells are capable of storing two bits per cell. Future roadmaps show 3 bits/cell and even 4 bits/cell as being possible for certain applications.

NAND memory must be used with error correction. Additional bits are provided within each memory block for the storage of error correction codes. Typically a BCH or Reed-Solomon code

Memory Type	Cell Size	Endurance
SRAM	$142F^2$	$> 1E15$
DRAM	$6F^2$	$> 1E15$
NAND	$4F^2$ - $2F^2$ *	10K-100K
PCRAM	$4F^2$	100K-1E6
MRAM	$40F^2$	$> 1E15$
* Multi-Level Cells		

Table 6.4: Memory types and characteristics.

will be applied to NAND blocks, allowing several errors to be present without risk of data loss.

NAND Flash memory is not capable of infinite read-write cycles (termed **endurance**). Most **Single-Level-Cell (SLC)** NAND is rated for 100K cycles with ECC. MLC NAND is usually rated for fewer cycles, even under 10K cycles.

The slow writes, block-oriented erase and programming, and the limited endurance of today's designs all make NAND unsuitable as a main memory replacement. As a replacement for, or alongside rotating magnetic media, NAND has great potential. Challenges do exist in scaling NAND gate and dielectric structures[113], but the industry is putting significant research efforts into the problems.

6.3.2.4 Alternative Memory Types

There are many types of memory that are being investigated in an effort to find replacements for SRAM, DRAM, and NAND Flash memories. The "Holy Grail" of memories would be one that is faster than SRAM, of unlimited endurance like DRAM, and both dense and non-volatile like NAND Flash. In addition, the cells should be compatible with existing CMOS logic, with a low-cost manufacturing process and consume near zero power. Of those memory types that have demonstrated some degree of commercial viability, few have shown that they are as economical or robust as today's memory leaders. Table 6.4 summarizes the cell sizes and endurance of some of the top memory contenders.

6.3.2.4.1 Phase Change Memory One of the most promising memory types on the near term horizon is **Phase Change Memory (PCRAM)**. Phase change memories operate by melting a small region of material through resistive heating, and then cooling under controlled conditions to either an amorphous or crystalline solid, which then exhibits two resistance states. The material used in the cell is a chalcogenide glass, and while the thought of melting glass may seem an unlikely memory technology, it may be viable. One of the factors in favor of phase change memory is that it actually improves as it gets smaller: As the melted regions are shrunk, the current required to melt the region is reduced.

Given the characteristics of phase change memories, it is most likely they will find use first as a NOR Flash replacement then perhaps as a NAND Flash replacement. Their currently projected endurance characteristics will keep them from main memory applications.

6.3.2.4.2 SONOS Memory **Semiconductor Oxide Nitride Oxide Semiconductor (SONOS)** memory cells are generally considered to be a natural extension of Flash memory technology. Rather than utilize a floating gate for the storage of charge an Oxy-Nitride-Oxide floating trap layer is used. It is anticipated that SONOS memory cells will be constructed that are equal to floating gate

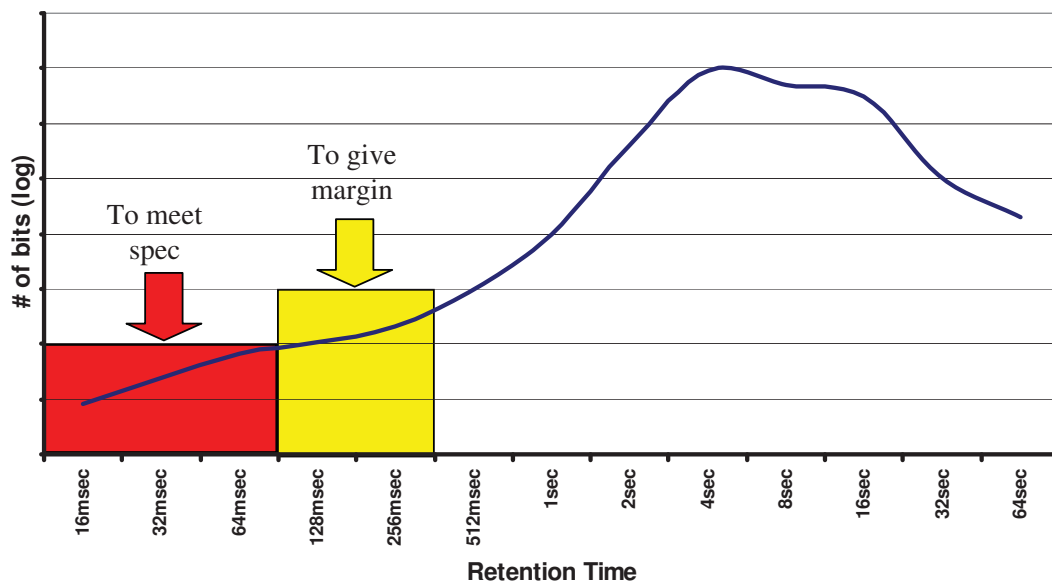


Figure 6.16: DRAM retention time distribution.

(FG) NAND cells. Endurance of SONOS cells has been observed to 10^7 cycles, which is on par with FG NAND. One additional feature of SONOS cells is that the programming voltage is lower, which is a definite advantage as the cells continue to shrink. SONOS cells also exhibit radiation hardness[156].

One area of research for SONOS and other memory technologies is in 3-dimensional stacking of memory cells. As scaling in X and Y become more challenging, integration in the Z-direction by stacking multiple layers of memory cells is showing some promise[70].

6.3.2.4.3 MRAM **Magnetic Random Access Memory (MRAM)** is based on the use of **magnetic tunnel junctions (MTJs)** as memory elements. MRAM is a potentially fast non-volatile memory technology with very high write endurance. One of the issues with MRAM is that it requires high currents to switch the MTJ bits. These write currents may be in excess of 5mA per bit and present a host of design challenges. This reason alone likely limits MRAM to use in smaller arrays in specific applications, such as rad-hard systems for space applications.

6.3.3 Main Memory Reliability - Good News

The DRAM used in today's computing devices has proven to be one of the industry's reliability success stories. In part, this reliability is due to the on-chip redundancy and how this redundancy is used to provide each memory chip with a maximum level of operating margin. While DRAM for computing applications generally specifies a 64msec refresh period, the processes used to build DRAM generally results in cells that retain a readable charge for several seconds.

Programmable redundancy was first suggested for DRAM as early as 1969; however, it was only implemented on a production basis with the 64Kbit and 256Kbit density generations in the mid-1980s. The programmability of these redundancy solutions has taken multiple forms, and today is split between **laser-trimmed fuses** and **electrical anti-fuses**. The most notable

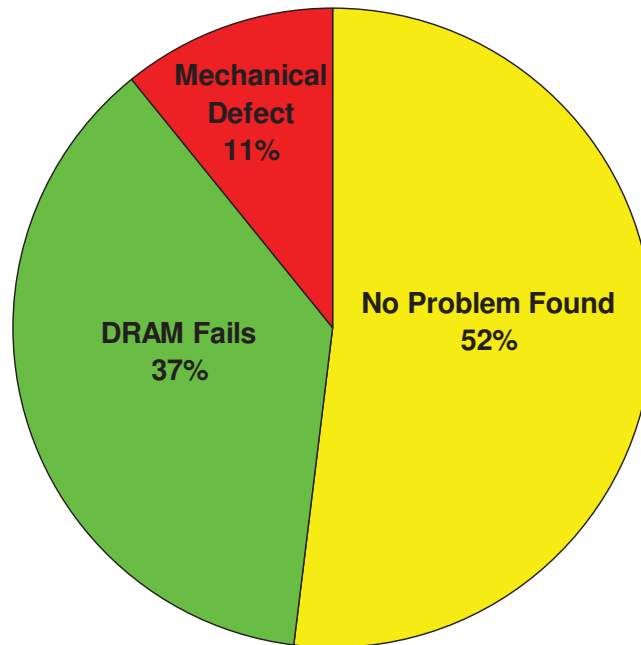


Figure 6.17: Memory module RMA results.

advantage of the electrical anti-fuse is that the arrays can be repaired late in the manufacturing process, even after packaging and burn-in.

Currently, DRAM manufacturers are using all available fuses to maximize the margin in the devices, as illustrated in Figure 6.16. It might be possible in the future to reserve a portion of the DRAM redundancy elements for use within an operational system such that field failures could be repaired in a live system. Clearly this would not be possible with laser-trimmed fuses. For electrical anti-fuses there are still challenges. No standards exist for accessing these fuses and even how these fuses interact with the memory array. Even within one manufacturers product portfolio there are often significant differences in fuse configurations between parts. “Super-voltages” need to be applied to appropriate interface pins which may not be compatible with other elements connected to those pins. Some engineering could resolve these issues although it is unlikely such a system would find its way into mass-market commodity DRAM.

6.3.3.1 Trends in FIT Rates

A **FIT** is defined as the failure rate for one billion operational hours. Existing mature DRAM products have shown tremendous improvements in FIT rates, even into single digits (less than 10 failures per billion hours). However, the sheer volume of memory, and number of potential memory devices, in at least the largest of Exascale machines will ensure that memory failure is an issue that must be dealt with. For example, if the memory FIT rate is 5 per memory device, and an Exascale machine has 0.1 Exabyte of memory made from 1 Gigabyte memory devices, the system could experience a memory failure every two hours. Clearly the Exascale machine must be designed with some resiliency for failures in the memory subsystem.

One common source of failures today is not accounted for in these FIT numbers - the memory DIMM sockets. The data in Figure 6.17 from a major manufacturer of memory devices shows the

distribution of **RMA (Reliability, Maintainability, and Availability)** results on analysis on reported failures in that company's memory modules. Of the 52% which indicate "No Problem Found" it must be assumed that issues with module sockets are responsible for a large portion of the returns.

While RMA data is typically indicative of early field failures, the memory socket problem may also have a long term aspect. Anecdotal evidence from a large support organization indicates that simply removing and re-inserting the memory modules cures nearly all memory-related problems.

6.3.3.2 Immunity to SER

Memory soft errors due to ionizing radiation are a common part of the collective experiences of most large system designers. As recently as 2003 at least one supercomputer was built without regard for soft error rates, and demonstrated its usefulness as a cosmic ray detector rather than as a computer. This system was subsequently dismantled and re-built with support for Error Correcting Code (ECC) memory with much better success.

While **soft error rate (SER)** is a bad memory (pun intended) for the industry, the good news is that, for properly designed devices and systems, SER for DRAM is but a memory and DRAM memory is largely without SER.

What makes DRAM SER resilient? Ionizing radiation may be viewed as discrete events at the silicon level. Such radiation has finite energy. DRAM cells, word lines, and bit lines are all highly capacitive. These forms of ionizing radiation simply lack the energy needed to affect the cell. Even when a cell would strike one of these nodes, the timing of the strike would need to be at just the right time to even be noticed. Furthermore, DRAM cells are so small that they individually represent a small target, but collectively represent a large capacitance in a unit area. If ionizing radiation strikes the silicon, there is a lot of capacitance in the area to absorb the effect. Certain nodes in the DRAM device could still remain sensitive to ionizing radiation, however the industry has been careful to avoid layouts that are such. This is particularly true of the sense amp area.

SRAM cells are not so fortunate. In order to allow increased operating speeds with reasonable power the SRAM cell uses very low capacitance nodes. As SRAM cells are quite large, $140F^2$, they present a very large target for ionizing radiation. To keep SER at reasonable levels SRAM arrays must employ careful layouts - such as ensuring that neighboring cells are addressed in different words - and must employ external error correction.

6.3.3.3 Possible Issue: Variable Retention Time

There is a seldom-discussed characteristic of DRAM that must be addressed due to the impact it could have on an Exascale system. This issue is the **Variable Retention Time**, or **VRT**, bit. Sometimes called "flying bits" the VRT bit is one that was first publicly acknowledged in 1987 by AT&T Bell Labs[16], and was confirmed by IBM[6], to exist in all known DRAM technologies and process nodes. The VRT bit is a memory bit that shows changes in its retention behavior, typically flipping its retention time between two values due to some external influence. This external influence might be temperature (packaging processes, reflow), stress (mechanical and electrical), x-rays (from inspection), or other influences.

6.3.3.3.1 Causes The causes of VRT bits are uncertain. It is likely that the mechanism has actually changed with different process generations. Today the most likely cause of VRT bits is the possible existence of a trap in or near the gate of the access transistor. The activation of the trap

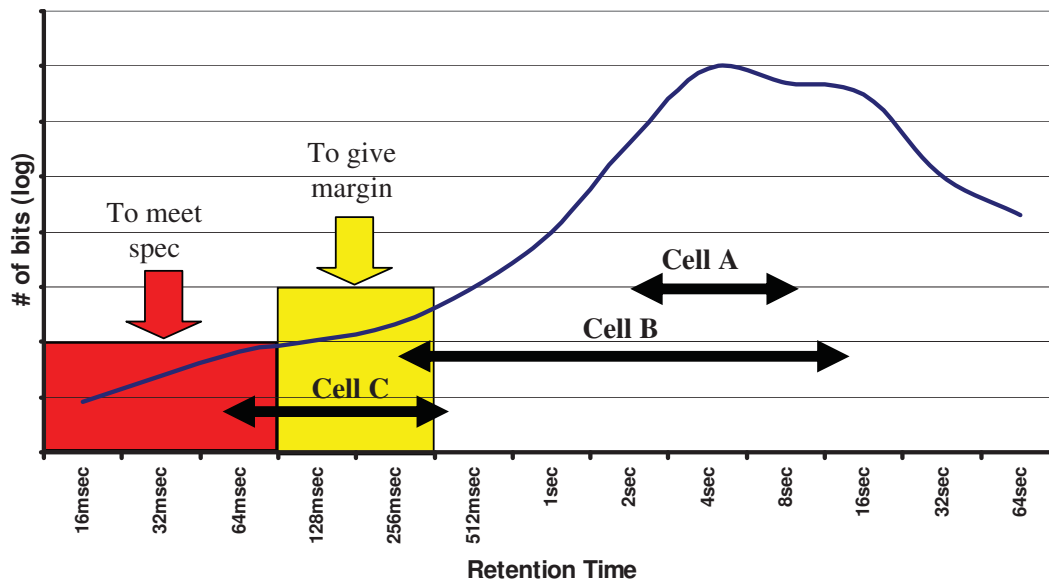


Figure 6.18: Variable retention time as it affects refresh distribution.

possibly affects the leakage of the access device, moving retention from long to short or from short to long.

6.3.3.3.2 Effects Consider the distribution of retention time for the billion-plus bits on a DRAM device. As previously discussed, redundancy in the array and fuses are used to remove short retention time cells from the array of cells seen by the system. Additional redundancy and fuses are used to provide a margin between the specified refresh time for the device and the refresh distribution of cells in the array. Now suppose the DRAM device is subjected to one or more of the VRT trigger events. It might be the DRAM being reflow soldered to a module PCB. It could be the injection molding package operation. It could be PCB inspection. If a cell is affected by this event and experiences a shift in its retention time to a time shorter than the refresh period seen by the device in the system that bit may appear to be bad. Of course another trigger event may cause the cell to return to its long retention time.

In Figure 6.18 a VRT shift in the retention time of Cell A or Cell B does not cause a noticeable change in memory behavior: all bits still look good. However Cell C shifts its retention time between an acceptable time and a time that is shorter than the system specified 64 msec. This cell will show up bad if it is in the short retention time state, and good in the long retention time state.

6.3.3.3.3 Mitigation Fortunately the frequency of VRT cells is low, and it is statistically unlikely that multiple VRT bits would show up in a single DRAM word. At the system level the addition of ECC is sufficient to remove the risks of VRT cells. As process technologies evolve, the industry continues to search for ways of limiting and reducing the prevalence of VRT cells in devices. VRT should remain as a concern however, as circuits continue to scale and as innovative packaging solutions are incorporated into Exascale systems.

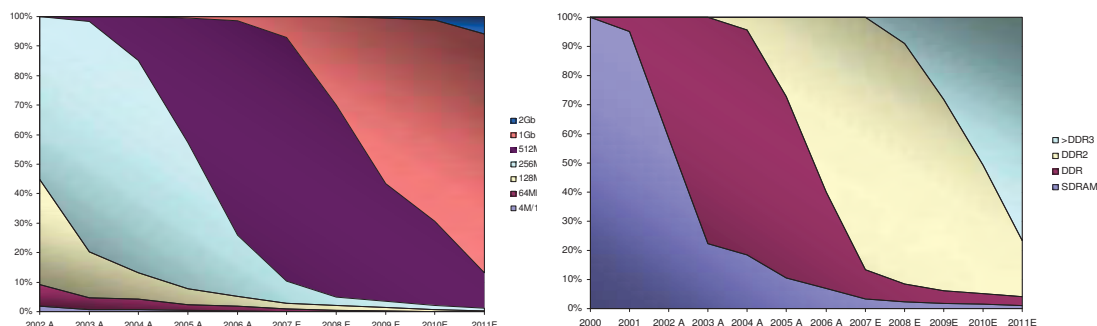


Figure 6.19: Industry memory projections.

6.3.4 The Main Memory Scaling Challenges

As shown in other sections of this report, the Exascale system demands unprecedented volumes of memory and that the memory must have both extremely high bandwidth and low latency. The sheer volume of memory constrains the power of individual memory devices. To achieve desired latency and bandwidth requires new architectures and interfaces between CPU and memory.

While commercial systems would greatly benefit from the types of architectures and interfaces discussed in this document, evolution of commercial memory is driven by committee (**JEDEC**) and may not serve the needs of such Exascale systems.

6.3.4.1 The Performance Challenge

Memory performance may be broken into two components: **bandwidth** and **latency**. Commercial DRAM evolution continues to improve interface bandwidth, while sacrificing memory latency. As the industry has moved from DDR to DDR2 and now to DDR3, the interface speeds have increased while the latency has also increased. The reason for this is straightforward. As the interface speed increases the memory chips must incorporate deeper and more complex pipelining logic due to the near constant access time for the array. Unfortunately the additional logic consumes more silicon area and more power. The challenge for the Exascale system will be to deliver the performance while simultaneously decreasing die size and power.

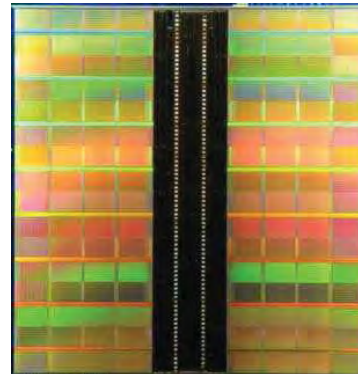
Commodity DRAM is forecast to continue density scaling with modest improvements in interface speed. The iSupply data[73] in Figure 6.19 show the projected trends for commodity DRAM. The Y axis in both cases is the percent of chips shipped.

6.3.4.1.1 Bandwidth and Latency DRAM can deliver higher bandwidth and lower latency. Figure 6.20 shows a die photograph of a reduced latency DRAM (RLDRAM) device. This device uses architectural changes to the sub-arrays to achieve faster array access times. There is a significant 40-80% die area penalty in making this type of change to the array. Additionally, this part supports a 36-bit wide interface, which has some additional die-size impact (more interface logic in the middle). If an Exascale system is to take advantage of lower DRAM latency and high bandwidth, a new architectural approach must be taken.

6.3.4.1.2 Tradeoffs There are some tradeoffs that can be made between bandwidth and latency. One tradeoff demonstrated in commodity DRAM is the use of additional I/O pipelining



(a) A Conventional DRAM



(a) A Reduced Latency DRAM

Figure 6.20: Reduced latency DRAM.

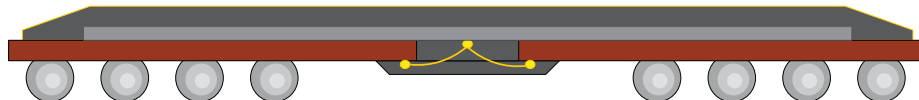


Figure 6.21: Center-bonded DRAM package.

to give bandwidth, at the expense of latency. Some non-standard DRAM architectures have even gone beyond commodity DRAM in this regard. This tradeoff has made sense for DRAM manufacturers as the peripheral transistor performance has been lagging behind that of logic processes. The additional pipeline stages become necessary to achieve the high bandwidth specifications. As in CPU architectures, the logic between pipeline stages is also reduced as the frequency climbs. Of course, an unintended tradeoff made when following this course is that the logic power and logic area of the DRAM increase.

One interesting study could be made of the possible power and performance results of an architecture which minimizes I/O pipelining logic, utilizing wide I/O buses, and instead utilizes some silicon area for improving latency and power through array segmentation.

6.3.4.1.3 Per-pin Limitations DRAM processes have been optimized for low leakage (increased data retention time) and high manufacturability. Low leakage transistors are not on par with high performance logic transistor performance. In addition, DRAM devices are generally optimized towards low pin counts for low cost packaging and low cost test.

Commodity DRAM has traditionally favored full-swing, single-ended signaling, although next generation DRAM may have low-voltage differential signaling. While differential signaling will likely double the signal bandwidth, it takes twice as many pins to deliver the data. Power will likely climb again.

6.3.4.2 The Packaging Challenge

Commodity memory packaging is driven largely by cost. A standard lead frame technology remains the preferred low-cost package for most applications. However, this is changing. In certain markets,

the need for increased density or improved signal integrity justifies additional packaging expense. The micro-FBGA packaging utilized in some server memory modules is one such example. Most commodity DRAM solutions rely on memory die with centrally-located I/O pads. I/O is limited to one or two rows of contacts that run down the center of the die. Figure 6.21 illustrates a cross section of a commodity micro-FBGA solution.

If the Exascale system requires memory with wider I/O, it is unlikely that existing packaging technologies will provide a viable cost-effective solution. Pin count, memory density and signal integrity likely drive the solution towards some sort of 3D die-stacking technology. There are many such technologies, but many are not suitable for DRAM. Wire-bonded solutions allow limited stacking as they typically require die periphery connections.

Existing commodity DRAM is already a highly 3D device. Cell capacitors, whether trench or stacked technology, are high aspect-ratio devices either constructed atop the silicon or etched into it. Capacitor surfaces are intentionally roughened to increase surface area. Access transistors are highly-engineered 3D structures optimized with unique channel profiles to reduce leakage.

In order to achieve densities required in certain applications, DRAM devices have been stacked at the package level for years. Only recently have stacked die-level DRAM devices been in high-volume production. Innovative packaging technology has enabled these solutions, but the industry is advancing towards more integrated stacking solutions that can be achieved at the wafer scale using semiconductor processing techniques.

Through silicon vias compatible with DRAMs are under development at most DRAM manufacturers. Interest in stacking DRAMs also comes from outside the DRAM industry as others look to integrate more power-efficient or high performance multiprocessor solutions.[16][80]

Some of the challenges with efficient stacking of DRAM are related to the existing 3D structure of the cells. To efficiently stack die requires die that are sufficiently thinned so that the through-wafer via etch may be done economically. However, the 3D structure of the cell limits the degree to which the die may be thinned. As previously noted, thinning of the die may also affect the refresh performance of the DRAM cells. Additional research is needed to overcome these challenges to develop a 3D stacking process that is high-yield and economical.

6.3.4.3 The Power Challenge

The Exascale system will face major challenges in the area of memory power consumption. Power in DRAMs come from two major sources: accessing the memory arrays, and providing bits off-chip. Commodity memory devices have only recently begun to address power concerns as low power DRAM devices have become standard for applications in mobile phones and portable electronics. Most often these DRAM devices have achieved power savings through modest process enhancements and architectural advances. In many cases the architectural advances have been towards improving the standby current, with a goal of improving battery life for portable consumer electronics systems. The Exascale system will demand a more aggressive approach to power management. Once again, this is due in large part to the sheer scale of the system. However, one point must be clearly understood: the DRAM cells themselves are very efficient. It is not the cells, but rather the interface to those cells that is the major consumer of memory power.

6.3.4.3.1 Module Power Efficiency Figure 6.22 illustrates the historical trend for power in commodity memory modules as a function of off-bit bandwidth. Each such memory module has multiple DRAM die that are driven together, and together provide system bandwidth. As can be seen, the power efficiency of memory, measured in mW/GB/s is improving at only a modest rate.

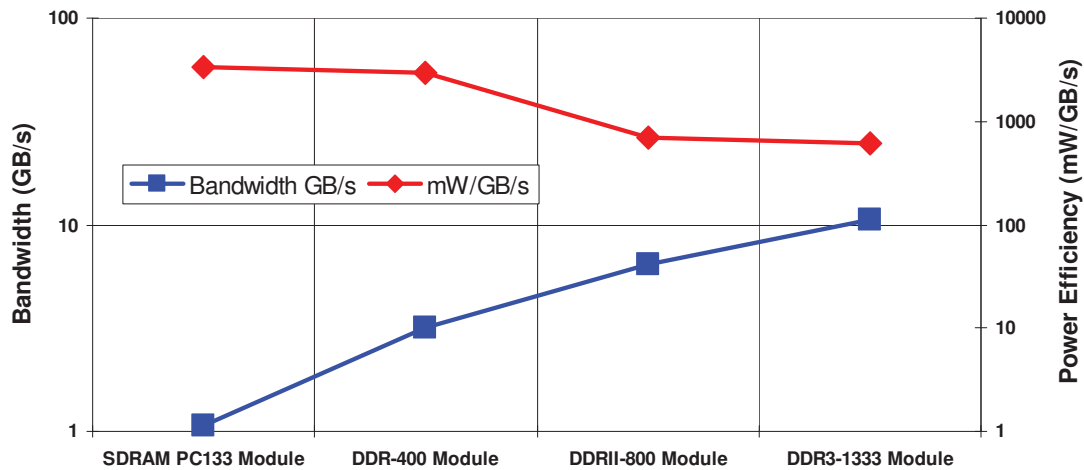


Figure 6.22: Commodity DRAM module power efficiency as a function of bandwidth.

6.3.4.3.2 Cell Power Memory power at the cell level is fairly easy to understand. A DRAM cell may simply be viewed as a capacitor which must either charge a bitline or be charged by that bitline. The DRAM capacitor is currently in the range of 25 femto Farads for most commodity applications. The bitline capacitance is considerably higher, typically several times that of the cell capacitance. If one assumes a total capacitance of 100 fF and a cell voltage of 1.8V then the energy to write or read a cell may be approximated as:

$$Energy = Capacitance * Voltage^2 = 100fF * 1.8V^2 = 81femtoJoules \quad (6.9)$$

This level of energy efficiency is never seen at the periphery of the memory device. A memory cannot exist without the row and column decoders required to access the desired cells. Steering logic is required to direct data from sense amps to I/O. All of these consume additional energy regardless of the type of memory cell technology employed.

Voltage scaling is slowing for DRAM as it is for logic. Figure 6.23 shows the trend in DRAM operating voltage, which is not expected to scale far beyond 1 volt. The possibility of reduced cell capacitance, device variability in the sense amplifiers and noise margin are all contributors to a slowing of voltage scaling.

An area of research worthy of study would be development of the technologies required to scale DRAM cells and periphery circuitry to lower voltage operation. Operation at 0.5V is, in theory, possible, but remains out of reach with current developmental paths. Such research could bear significant commercial dividends as well. By some estimates[87], data centers consume 14% of the country's power production growth. The same study estimates memory power consumption to be 27% of the data center's power load.

6.3.4.4 Major Elements of DRAM Power Consumption

As shown in the previous section, DRAM cells are actually quite efficient by themselves. However, there remains room for improvements to the cell. Voltage scaling must be extended and is not supported by current roadmaps and research. Voltage scaling must be extended to the periphery of the DRAM array as well, a task that also remains out of the reach of current roadmaps and

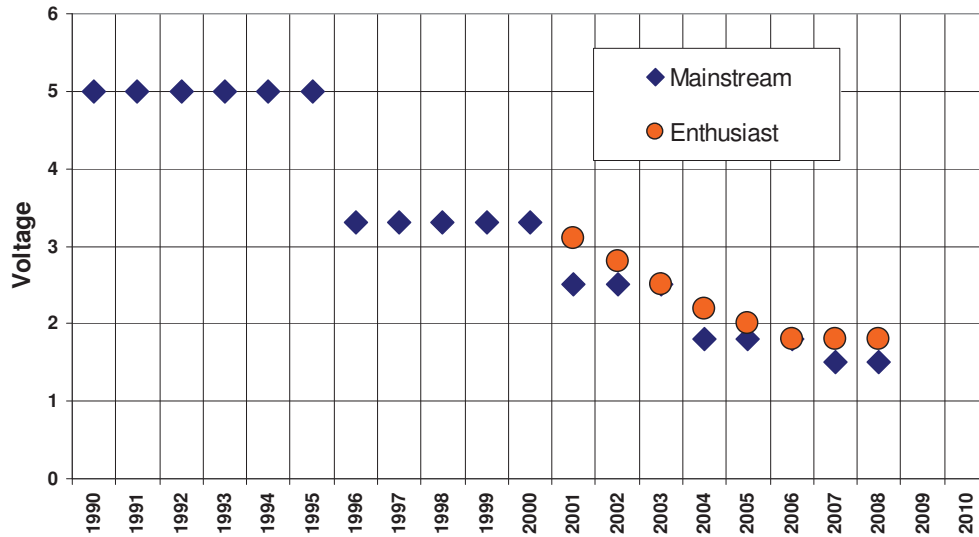


Figure 6.23: Commodity DRAM voltage scaling.

DDR2-400 IDD Specifications		
Operation	Symbol	Current
Idle	IDD2P	7ma
Refresh (Burst)	IDD5	280ma
Precharge	IDD2Q	65ma
Activate	IDD1	110ma
Read	IDD4R	190ma
RWrite	IDD4W	185ma

Table 6.5: Commodity DRAM operating current.

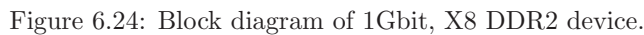
research. In order to understand the impact of DRAM periphery in the overall memory power picture it is helpful to study current devices and trends.

6.3.4.4.1 DRAM Operating Modes DRAM operation may be broken into six operations: **Idle**, **Refresh**, **Precharge**, **Activate**, **Read**, and **Write**. For the purposes of this discussion on Exascale systems we may ignore the power down modes that are often used in notebook and desktop computers in their Standby operation. For a state of the art DDR2-400 memory device, typical operating current for each of these operations is shown in Table 6.5, along with their common notation.

As Idle is of little interest, and Refresh is a small percentage of overall operation we will focus our attention on the Precharge, Activate, Read and Write operations.

Before a DRAM cell can be read, the bitlines associated with that row must be brought to a $V/2$ level, where V is the operating voltage of the DRAM array. This is the Precharge operation and is really just a preparation for a Read or Refresh operation.

The Activate operation is the opening of a memory row for a read or write operation. During the Activate operation the row decoders select the appropriate wordline and drive that wordline to



Similarly, a write operation begins with a Precharge and Activate. As with the Read operation, the row decoders must decode and drive the appropriate wordline with the pumped voltage. The written data is driven into the array by the sense amplifiers. The bits in the row that are not being written are refreshed in this same operation.

Finally, the “I/O Gating and DM Mask Logic” block is greatly simplified. Helper flip-flops store the output of the sense amplifiers. A critical speed path in the architecture on most designs is the gathering and steering of data to and from the multitude of sub-arrays. It is a complex logic element that incorporates most of the fastest (and therefore most power-hungry) transistors in the

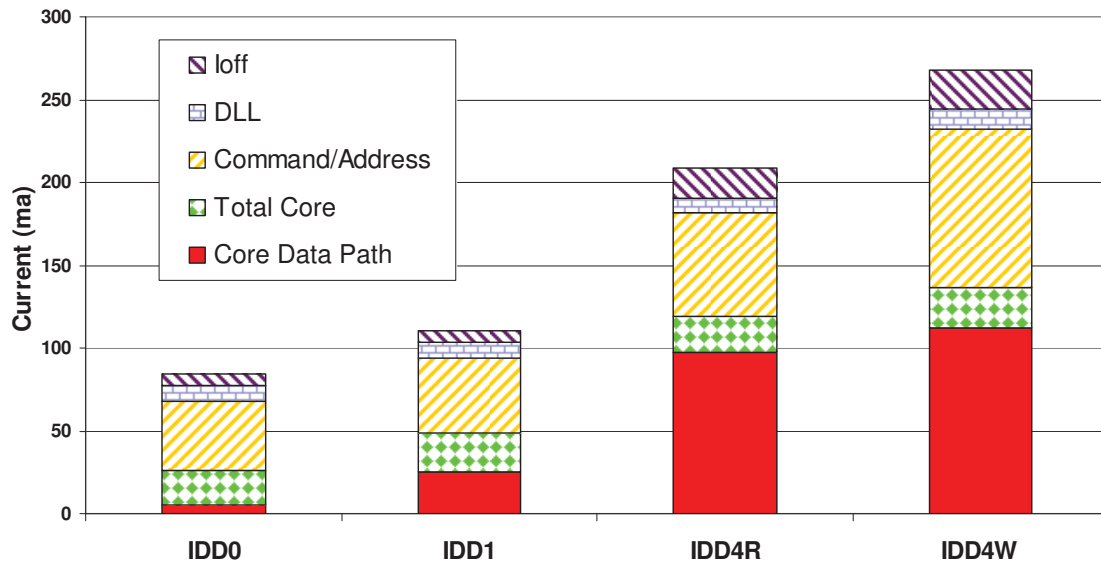


Figure 6.25: DDR3 current breakdown for Idle, Active, Read and Write.

device. Also included in this logic are structures for device testing.

Figure 6.25 tallies the current drawn in each of the major sections of a commodity DDR3 DRAM device as a function of operation mode. Multiplying this by V_{dd} and by the percent of time that the chip is performing that type of operation yields power. The numbers presented are similar to those from a DDR2 device as shown in Figure 6.24.

This figure illustrates what has been said before: The total core power is dwarfed by the power of the surrounding circuitry when the device is operating.

6.3.4.4.3 Power Consumption Calculations An accurate memory power calculator is beyond the scope of this report. Such technology does exist for today's commodity DRAM devices and the reader is invited to experience one[106]. Tools for analyzing SRAM power and allowing architecture optimization have existed for many years. One such tool, CACTI, shows some potential and could be enhanced to support DRAM architecture analysis and the advanced ITRS roadmap[158][98][145].

For the purposes of this report the needs of the Exascale machine must be considered, especially a data center-sized system. It is safe to say that the commodity DRAM roadmap does not fulfill the needs of the Exascale machine. Power consumption is simply too great for the type of memory footprints under consideration. For example, given that a DDR3 chip today consumes just over 600 mW/GB/sec, the power budget for an Exascale data center machine requiring 1 EB/sec of main memory bandwidth would be just over 600 megawatts, which is simply not viable.

But step back for a minute and consider the memory at the cell level. As previously stated, DRAM cells are not inefficient, with cells requiring only about 80 femto Joules for switching. If one considers just a memory array delivering 1 EB/sec bandwidth, these DRAM cells consume only about 200 Kilowatts of power! (based on a future 1 volt, 25fF cell cap). When bitline capacitance is added, the number becomes 800 Kilowatts, based on a cell to bitline cap ratio of 1:3[79].

In current DRAMs, one must consider the power consumed across the entire row. In typical

commodity DRAM, 8K bits are accessed for any read or write operation. If the system is able to use additional data from the 8K row then the row may be left “open” for sequential accesses. However, in practice there is usually little locality in memory references, and it is most efficient to “close the row” to prepare for another random access to the device. What is the power penalty paid here? For a commodity 8-bit wide DRAM device that bursts 4 bytes to fill a cache line, and with an 8K bit row, the power consumption jumps to over 51 Megawatts for 1 EB/sec. (Note that the row access rate is now reduced to 1/4th of the previous due to the 4-byte burst at the interface.) Clearly, over-fetching of unused data by opening an entire DRAM row is the most significant power problem with scaling commodity DRAM architectures to the Exascale machine.

Driving the wordline/indexwordline can also consume substantial power. Here, the voltage is pumped, and the line is long, resulting in significant driving energy. If we assume the wordline capacitance is 3pF for the entire 8K row, and that a word line must be driven to $2 * V_{dd}$ at a rate 1/4th that of the data rate (burst transfer length of 4), the power consumption of the aggregate Exascale memory wordline is about 1.8 Megawatts for the same future 1V memory cell. Of course, this does not include any of the pumps, level translators or row decoder power.

It should be clear that the DRAM memory cell power is not the limiting factor for the power consumption of the Exascale memory subsystem. Other technologies may be proposed for the memory cell, but these will result in the same power-hungry memory subsystem if the periphery of the array is required to perform the same types of operations.

Future research directions to enable the Exascale memory subsystem must include analysis of techniques to reduce power consumption around the memory array.

6.3.5 Emerging Memory Technology

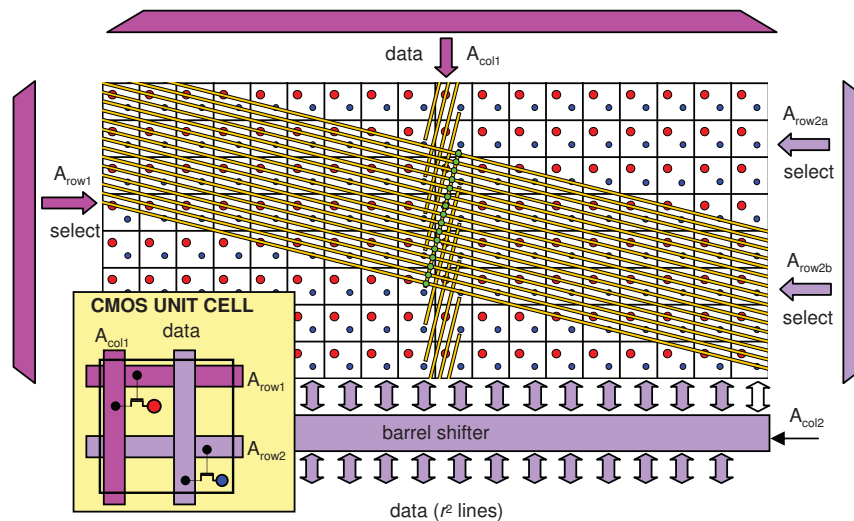
There is currently a significant effort world wide to develop new **non-volatile random access memory (NVRAM)** that can complement or replace existing memory and storage technologies. While many different NVRAM contenders are rising to this challenge; at this stage, it does not look like any will replace an existing technology outright. However, there may be applicability with new system architectures that place a layer of NVRAM between DRAM and magnetic disk storage to buffer the latency and bandwidth gaps between present memory and storage technology options. Driving this bandwidth gap wider (and thus making such architectures more valuable) are two trends: (i) off-chip access requires more and more clock cycles and (ii) significant nonvolatile storage is demanded by an increasing number of applications.

Today the highest density emerging memory technologies will meet this need by combining a dense crosspoint memory array with a fast, nonvolatile crosspoint device. A crosspoint memory offers the highest possible $4F^2$ cell density in a single layer, and most variations offer the possibility of significantly improving over this by being able to stack many layers of crossbars on top of a single addressing and control layer. Several device technologies may offer the needed combination of high bandwidth and low-latency nonvolatile electrical switching. These technologies include phase-change PCRAM, ferro-electric Fe-RAM, magnetic MRAM, and resistive RRAM[29][51][14].

Of these, FeRAM and MRAM devices are currently the most mature, with 1-4 Mb chips available for niche applications now, but expansion into large-scale computing environments has been hampered by poor device scaling (these technologies are limited in size by super para-electricity and super para-magnetism, respectively, which are made worse by the inevitable high temperature operating environments of an Exascale system), complex device structures and large fluctuations in corresponding behavior, and high power demands.

Phase change RAM are also power hungry, and show mechanical device instability over time.

Resistive RAM devices based on various metal sulfides or oxides, although less mature at present,



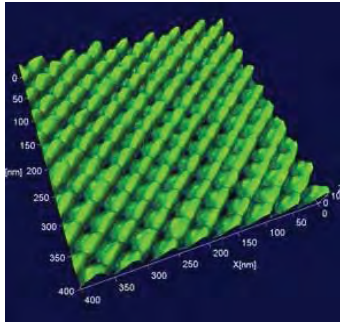
Addressing nanoscale bits in a particular memory segment with four CMOS decoders in ‘CMOL’ memory. The figure shows only one (selected) column of the segments, the crosspoint nanodevices connected to one (selected) segment, and the top level nanowires connected to these nanodevices. The nanowires of both layers fill the entire array plane, with nanodevices at each crosspoint. The inset shows a simple schematic of the CMOS cell structure for this memory.

Figure 6.26: Nanoscale memory addressing.

offer a compelling combination of relatively high speed/low latency (tens of nanoseconds), non-volatility (months to years) and low energy (2 pJ per bit in present unoptimized devices) switching, which could meet checkpoint demands and might even shift the requirements for DRAM for Exascale computing systems. At present, the metal oxide systems are receiving the most attention, since they are closely related to materials already found in today’s integrated circuit fabrication facilities (especially hafnium dioxide) and thus the amount of process work needed to incorporate them onto chips and the worries of incompatibility with existing fabs is minimal.

One of the primary problems with crosspoint NVRAM is the **multiplexer/demultiplexer (mux/demux)** needed to read and write information into the memory[28][56]. There have been several schemes introduced to enable the transition from scaled CMOS to nanoscale crossbars, including the use of coding theory to design defect- and fault-tolerant demuxes. The most ingenious scheme was proposed by Strukov and Likharev[94], who showed that it is possible to layer an extremely high density crossbar memory on top of a lower density mux/demux and still achieve complete addressability with defect- and fault-tolerance (see Figure 6.26). They simulated the bandwidth for such systems and determined that 1TB/s for a 10 ns read time with ECC decoding was achievable as long as the array is large enough that all overheads are negligible. The major challenge in this case would be to transmit the data off of or into an NVRAM package at the data rate that it can handle - this may actually require photonic interconnect or close proximity to DRAM to achieve.

To date, the highest density NVRAM structure that has been demonstrated is a 100 Gbit/cm² crossbar that was fabricated using imprint lithography, Figure 6.27[76][77]. Since this structure is made from two layers of metal nanowires that sandwich a layer of switchable dielectric material, this technology is well suited to stacking multiple crossbars on top of each other. Since the storage



An Atomic Force Microscope (AFM) topograph of a defect-free region in a 17 nm half-pitch nanowire crossbar fabricated by imprint lithography. This corresponds to a local memory density of ~ 100 Gbit/cm². Using the CMOL memory demultiplexing scheme of Strukov and Likharev, this crossbar could be placed over a two-dimensional CMOS array for reading and writing. Multiple crossbars could be stacked on top of each other to achieve even higher bit densities.

Figure 6.27: Nanoscale memory via imprint lithography

Year	Class	Capacity (GB)	RPM	B/W (Gb/s)	Idle Power(W)	Active Power (W)
2007	Consumer	1000	7200	1.03	9.30	9.40
2010	Consumer	3000	7200	1.80	9.30	9.40
2014	Consumer	12000	7200	4.00	9.30	9.40
2007	Enterprise	300	15000	1.20	13.70	18.80
2010	Enterprise	1200	15000	2.00	13.70	18.80
2014	Enterprise	5000	15000	4.00	13.70	18.80
2007	Handheld	60	3600	0.19	0.50	1.00
2010	Handheld	200	4200	0.38	0.70	1.20
2014	Handheld	800	8400	0.88	1.20	1.70

Table 6.6: Projected disk characteristics.

is non-volatile, there is no static power needed to hold or refresh the memory, and thus the thermal issues of stacking multiple memories are also minimized.

Defect tolerance in such devices is a major concern, with a growing effort to define fault tolerance mechanisms and project potential characteristics[9][33][95][141][142][143][163].

6.4 Storage Memory Today

This section discusses the technologies from which mass store memory used to implement scratch, file, and archival systems. Clearly, the major technology today is spinning disk, although there are several others emerging.

6.4.1 Disk Technology

Rotating magnetic disks have been the major technology for scratch and secondary storage for decades, so it was deemed important in this study to understand whether or not it can continue in that role into the Exascale regime, and discussions were held with leading disk firms, along with compilations of relevant historical data[2].

Disks have several major properties that need to be tracked: capacity, transfer rate, seek time, and power. Table 6.6 lists projections of these properties for three classes of disk drives:

Consumer: high volume disks where capacity and cost is paramount.

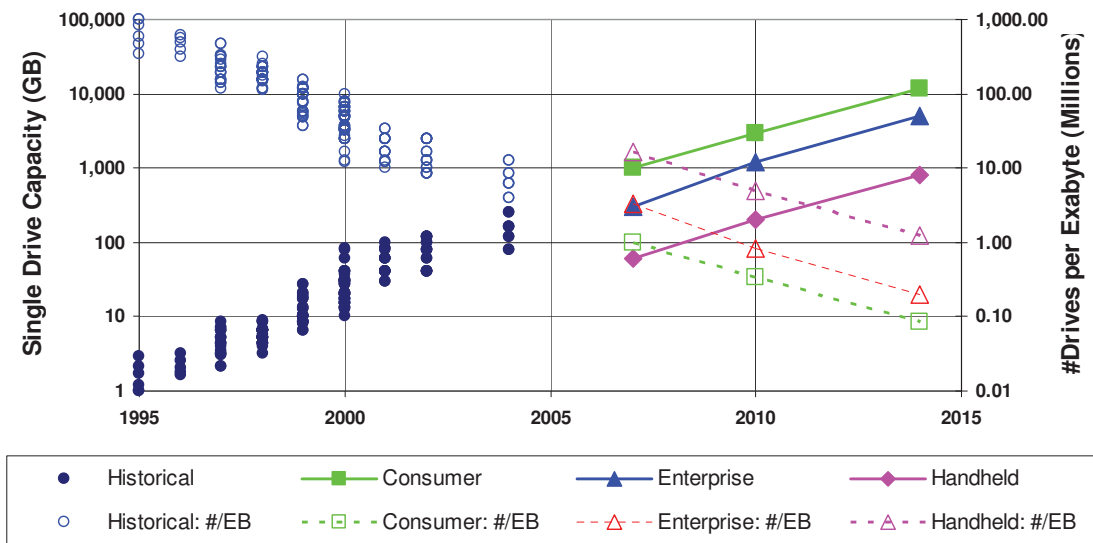


Figure 6.28: Disk capacity properties.

Enterprise: disks where seek and transfer time is paramount.

Handheld: disks where power and small size for embedded applications are paramount.

6.4.1.1 Capacity

To put these projections in perspective, Figure 6.28 graphs both historical and projected capacity. As can be seen, 10X growth over about 6 year periods seems to have been the standard for decades. Assuming that the basic unit of secondary storage for data center class systems is an exabyte, then depending on the type of drive, between 83 thousand and 1.3 million drives of 2014 vintage are needed per exabyte. Consumer drive technology, with its emphasis on capacity, requires the fewest drives, and handhelds the largest.

Any additional drives for ECC or RAID are not counted here, so these numbers are optimistically low.

Also not studied here is the actual physical volume needed for such drives.

6.4.1.2 Power

Another major parameter is power. Figure 6.29 projects the active power in MW for an exabyte of disks of each class. In 2013-2014, the range is between 0.8 and 3.8 MW, with enterprise class taking the most power (bigger drive motors to run drives faster).

We note that the power numbers here are for the drives only; any electronics associated with drive controllers needs to be counted separately.

Again ECC or RAID is not considered, so real numbers would be higher.

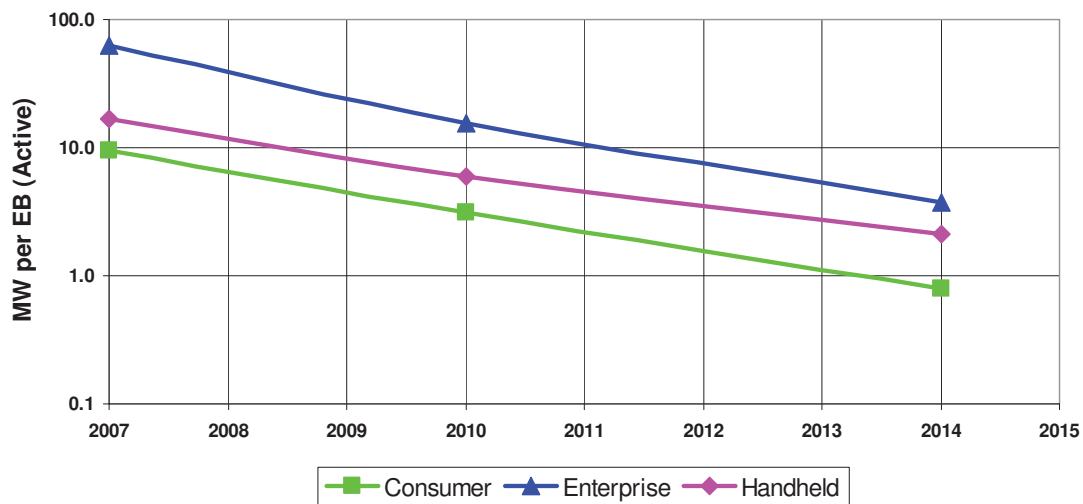


Figure 6.29: Disk power per Exabyte.

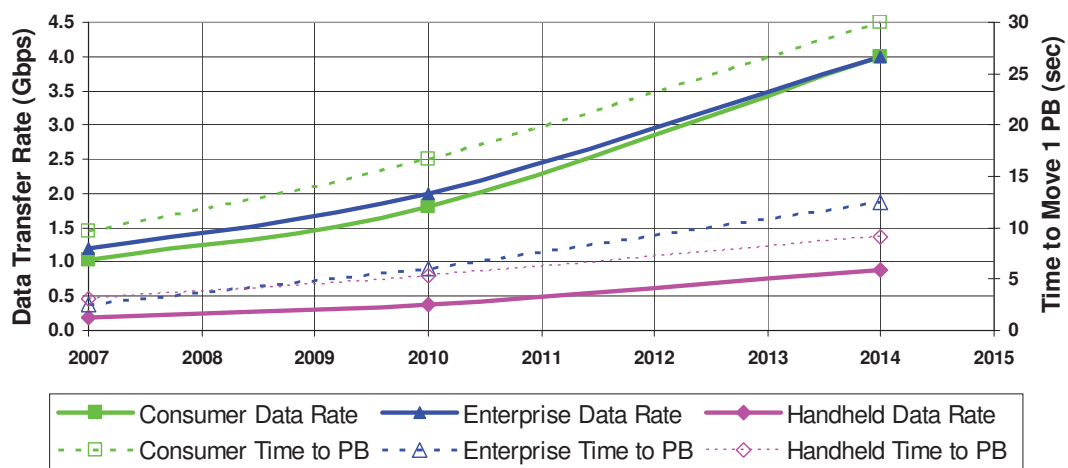


Figure 6.30: Disk transfer rate properties.

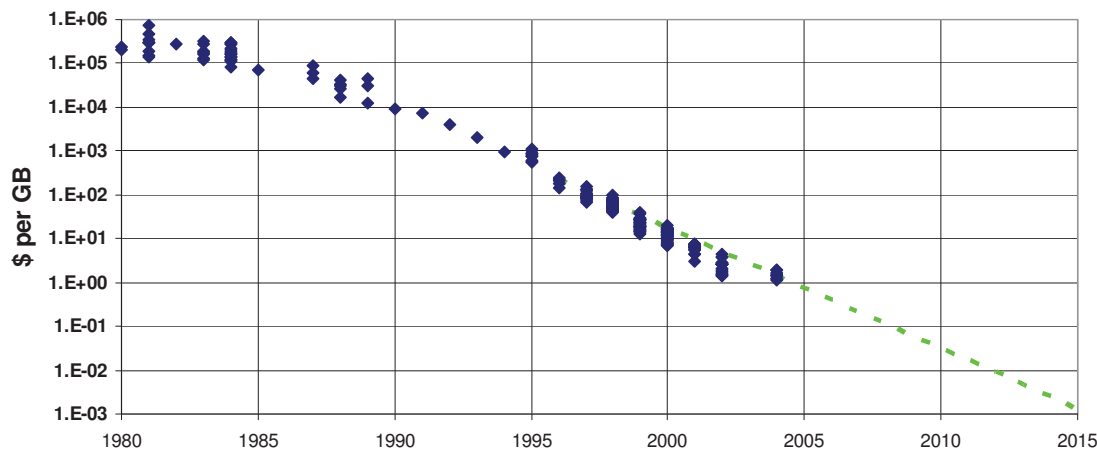


Figure 6.31: Disk price per GB.

6.4.1.3 Transfer Rate and Seek Time

The transfer rate of a drive provides one part of the overall bandwidth capabilities of a single drive. This transfer rate is defined as the maximum data rate that data can be streamed from a drive, given little or no head seeks. Achieving such rates requires data to be transferred in large (multi-MB) segments, and to be physically recorded in optimal places on a drive so that when one block is read, the next block to hold the next relevant chunk of data is directly under the disk heads.

For each of the three classes of drives, Figure 6.30 provides projections of the maximum transfer rate under the above conditions.

Of course, in real systems the number of seeks and the average seek time become a dominating consideration, especially for file systems with many small files. In general, the average seek time is proportional to the rotational rate. Unfortunately, as shown in Table 6.6 this rate seems essentially flat for all classes of machines, meaning that future drives of any class will not respond to seek requests any faster than they do today. This will become a huge problem, especially for searching directories and accessing many small files.

6.4.1.4 Time to Move a Petabyte

Also included on Figure 6.30 is an estimate of how much time it would be required to move one PB of data, assuming that we were able to perfectly stripe the data across the number of drives needed to contain an exabyte (from Figure 6.28), and that each such drive transferred data at the maximum possible rate. This transfer time increases even as the transfer rate of an individual drive increases because the number of drives over which the assumed exabyte of data is striped is decreasing. This is a very relevant number in discussions related to scratch disks and time to checkpoint data from memory to disk.

This is also again an optimistic minimal number, with no consideration for problems with drives out of sync, skew between the data arrival from the individual drives, or seek times.

Also again ECC or RAID is not considered, so one would expect that additional time would be needed to read out and perform any needed error detection or correction.

6.4.1.5 Cost

Although not a direct metric for this study, cost is still a consideration. Figure 6.31 gives some historical data on price per GB of capacity. To get to an exabyte, we would change \$1 from this figure to \$1 billion. The reduction rate in something in excess of 10X per 5 years, leading to a predication of a few \$10 millions for an exabyte in 2015.

Again neither RAID, controllers, nor interconnect cables are included in these estimates.

6.4.2 Holographic Memory Technology

Holographic memory refers to the use of optical holograms to store multiple “pages” of information within some storage medium. Recording is done by splitting a light source into two coherent beams, passing one through an image, and then recombining them on some photosensitive storage material, typically a photopolymer of some sort[39]. The recombination causes interference patterns, and if the intensity of the light source is high enough, the interference pattern is stored in the material. Readout is achieved by shining just the original reference light source through the material, and then detecting the resultant image.

For many materials, changing the angle of the light beams permits multiple holograms to be stored in the same material.

Given that the original data is an “image,” such holographic storage systems are “page”-oriented memories - reading and writing are in units of “images.” If an image is a “bit-pattern,” then in terms of a digital storage medium, a holographic memory is said to be a **page-oriented memory**. Typical sizes of such pages seem to be around 1 Mbit of data.

Two forms of such memories have been demonstrated to date: one based on 3D cubes of storage material, and one based on disks. In 1999, a breadboard of a 10 GB non-volatile cube was demonstrated which when combined with the required optical bench required 154in³[27]. More recently, a commercial product³ has demonstrated a drive with removable optical disks with 300 GB capacity with an overall form factor of about 700in³, a seek time of 250 ms, a transfer rate of 20 MB/s, 1.48Mb pages, and page write times of about 1 ms. The storage material in the latter demonstrated 500 Gb per in²[10], which is perhaps 10X the current density of hard disks, and about 60 times the density of today’s DVDs. This density is what makes the technology of interest, especially for archival storage.

In comparison, current DRAM chip density is around 84 Gb per in², and flash is about 2-4X that. Of course, this is density and not dollars per Gbyte, which today favors the spinning medium by a significant factor.

If bits per cubic (not square) inch are the metric, then disk drive technology seems to hold a significant advantage – a current commercial drive⁴ fits 1 TB in about 24in³, or about 100X denser. Even silicon has an advantage, especially as we find better 3D chip packing. The current holographic memories take a real hit on the volume needed for the optics, and unless techniques that reduce this by several factors (perhaps by sharing lasers), this is unlikely to change.

In terms of seek times (a key parameter when doing random reads), current disks are in the 8-9 ms range, significantly faster than the early commercial offering. Again, this is a first-of technology, and significant advances are probable, but there is still significant ground to make up.

³InPhase Tapestry 300r; <http://www.inphase-technologies.com/downloads/2007-11PopSciAward.pdf>

⁴Seagate Barracuda ES.2 http://www.seagate.com/docs/pdf/datasheet/disc/ds_barracuda_es.2.pdf

6.4.3 Archival Storage Technology

As discussed in Section 5.6.3.3, archival storage is experiencing both huge capacities, rapid growth, and problems with metadata for huge numbers of files. Today, storage silos and tape farms of various sorts are keeping up with the 1.7-1.9 CAGR of current installations, but it is unclear whether they will be able to make the jump to Exascale, especially for the data center class of systems which is liable to start off with 1000X of Petascale.

Besides the capacity issue, the metrics raised in [55], especially for “scanning” an archive as part of advanced data mining applications, focus attention on the need for dense enough and fast enough storage to hold not the data but the metadata that defines and controls the actual data files, especially when the potential for millions of concurrent accesses is present. Such data is read-mostly, at high speed and low power. It may be that some of the emerging solid-state storage mechanisms, such as a rearchitected flash, may very well become an important player in designing such systems.

6.5 Interconnect Technologies

The term **interconnect** revolves around the implementation of a path through which either energy or information may flow from one part of a circuit to another. Metrics involve both those that represent “performance” and are to be maximized, as in:

- **current flow:** as in when implementing power and ground delivery systems.
- **signalling rate:** as in the maximum rate that changes at the input of an interconnect can be made, and still be detected at the other side.
- **peak and sustainable data bandwidth:** as when digital information is to be transferred reliably via signalling.

and metrics that are to be minimized, as in

- **energy loss:** either during the transport process (resistive loss in the interconnect), or in the conversion of a bit stream at the input into a signal over the interconnect and the conversion back to a bit stream at the other end.
- **time loss:** where there is a latency involved between the launch of the information (or energy) at one end and its reception and conversion back into a useable form at the other end.

The challenges for Exascale systems lie almost wholly on those metrics associated with information transfer, namely latency, data bandwidth, and energy loss, and our emphasis here will be on the metrics of data bandwidth (measured in gigabits per second) and on energy loss (measured in pico Joules per data bit transferred).

Further, these challenges exist at multiple levels:

- on chip: over a distance of a few mm,
- chip-to-chip: where the chips are in very close coupling as in a stack,
- chip-to-chip: where the chips are further separated on some sort of substrate such as a PC board.
- board-to-board within a single rack,
- and rack-to-rack.

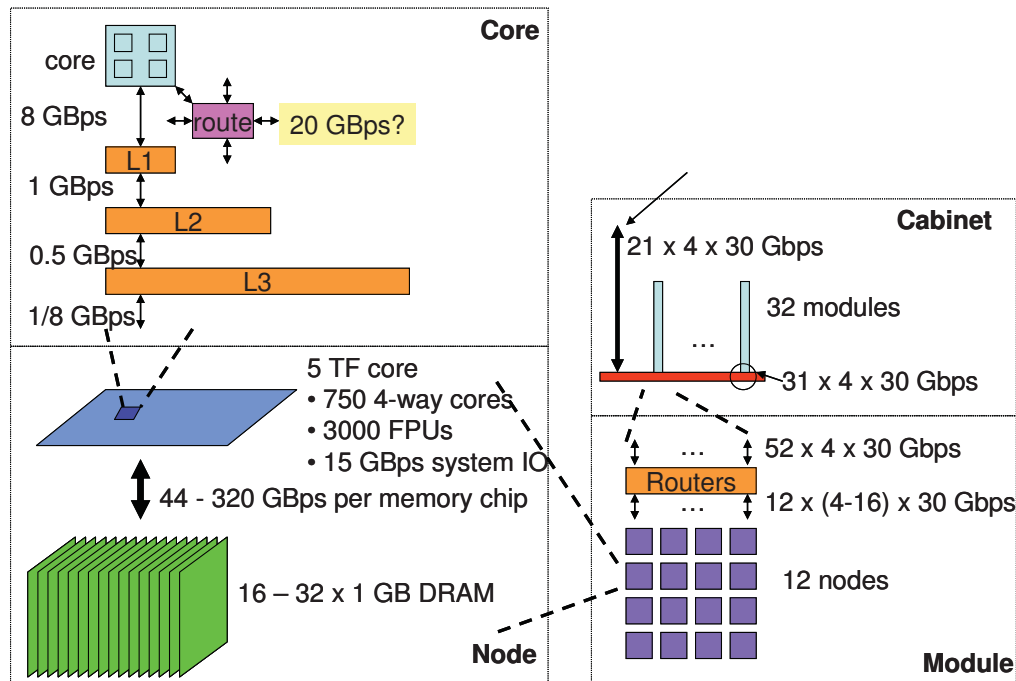


Figure 6.32: Interconnect bandwidth requirements for an Exascale system.

6.5.1 Strawman Interconnect

The first three types of interconnect are clearly relevant to all three classes of Exascale systems, “board-to-board” is relevant primarily to departmental and data center classes, and “rack-to-rack” primarily to data center scales. For discussion, Figure 6.32 summarizes possible interconnect bandwidth requirements for an Exascale data center system, as taken from the aggressive strawman presented in Chapter 7.3. The following paragraphs then discuss what was assumed for the aggressive strawman design in each of the above categories of interconnect, and set the stage for the discussion of alternatives in the following sections:

6.5.1.1 Local Core-level On-chip Interconnect

These include intra-core and local memory access. These lines are at most a few mm long and are point to point buses. At full swing with today’s technologies, these consume 110 fJ/bit/mm, and at reduced swing, 18 fJ/bit/mm. Reduced swing circuits have the disadvantage of requiring an amplifier receiver so the signal can be returned to full swing. A lower power receiver solution is to use a clocked receiver, for example a sense amp style circuit, but this adds latency depending on the clock phase used. Using low-swing signaling, L1-L3 memory bandwidth only consumes 3 W and has negligible area impact.

6.5.1.2 Switched Long-range On-chip Interconnect

The strawman does not discuss on-chip routing. However, it is plausible that an Exascale computing chip would benefit from a low-latency on-chip router. For example, the Intel Teraflop research

vehicle emphasizes on-chip routing [150]. Alternatives including using shared memory, and/or supplementing routed interconnect with switched interconnect. The implications of switchable routing will be discussed below.

6.5.1.3 Supporting DRAM and CPU Bandwidth

Large amounts of DRAM bandwidth are required. The suggested sustained bandwidth of 320 GBps per GB chip is about two orders of magnitude more than DRAMs provide today. At 30 Gbps per wire pair, 170 data pins per memory would be required. At 10 Gbps, 512 pins would be required. While the lower line rate would reduce the complexity of the SerDes, it would, in contrast, increase the packaging requirements. Assuming 16 GB of total memory and 30 Gbps per pair, the CPU would require 2,720 memory data IO pins. Assuming a 32-bit address space, 64 differentially connected address pins would be needed per memory. Add in 16 pins for control, gives a total of 250 pins per memory chip, or 4,000 pins for the CPU. Add in 4-16 pairs (12-40 pins including control) for the assumed data IO, and 2,000 power and ground pins for power delivery and integrity, gives a total of 6,000 pins on the surface.

6.5.1.4 Intramodule Bandwidth

In the strawman, each module is assumed to contain 12 nodes and 12 router chips (drawn here as one router). At 12-40 pins per node, the router has to handle up to 480 I/O connections to the nodes. It also has to handle ~500 I/Os to the board.

6.5.1.5 Intermodule Bandwidth

The equivalent of the system backplane or midplane has to be able to manage 32 blades, each with ~500 30 Gbps I/O in a point-to-point configuration.

6.5.1.6 Rack to Rack Bandwidth

In the strawman of Section 7.3, each rack is connected to every other rack by an unswitched 4x30 Gbps bundle (e.g. 8 wires carrying differential traffic or perhaps optical fiber). The strawman has a router at each module level so that there are no more than two router chips between any two CPU chips (neglecting any on-chip routing). Packet routing is assumed, with packet sizes starting at 64 bits.

In general it is accepted that fully routable packet-style routing is needed for a large scale computer that will be suitable for a wide range of applications. However, several large scale applications do display regular inter-process interconnect patterns. Thus, it is possible, but not verified, that additional performance (i.e. more efficient use of bandwidth) might be gained from the addition of a circuit switch function. For example, some applications map well onto a grid architecture, while others map well onto an architecture that reflects how operations are mainly formed on the main diagonal(s) of a matrix. However, given the limited total power budget, adding a circuit switch function could only be done by reducing CPU or memory, which would be an overall win only if the remaining resources are more efficiently used with circuit-switched communication. A performance analysis would have to be done across a broad range of applications before this could be deemed worthwhile.

6.5.2 Signaling on Wire

Classically, virtually all signalling within computing systems has been done by current transfer through wires. Within this category, there are two major circuit variants: **point-to-point** and switched. Each is discussed in a separate section below.

Note that **busses**, where there are multiple sources and sinks electrically tied to the same wire at the same time, are not addressed here - the aggregate capacitance of such interconnect makes them rather power inefficient, at least for longer range signalling.

6.5.2.1 Point-to-Point Links

Point-to-point interconnect occurs when there is a well-defined source transmitter for the data and a well-defined receiver, and the two circuits do not switch roles. Within this category, there are three variants relevant to Exascale: on-chip, off-chip, and switched. Each is discussed below.

6.5.2.1.1 On-Chip Wired Interconnect At the 32 nm node, we estimate a line capacitance of 300 fF/mm. With a 0.6 V power supply, **full swing signaling** gives a signaling energy of 110 fJ/bit-mm. Many schemes have been proposed to reduce interconnect power through voltage scaling. These schemes require an amplifier at the receiver to amplify the swing back, with some additional power needed. Alternatively a clocked sense can be used for a lower power receiver, with the implication that the latency includes a crossing of a clock phase boundary.

When using **lower swing interconnect**, two benefits arise. The first is reduced power consumption. A swing of 0.1 V reduces the power down to 18 fJ/bit-mm. A number of schemes have been demonstrated, and it is not necessary to distribute a 0.1 V supply. The second benefit is increased range without repeaters. Simple equalization schemes can be used instead. Repeater-less ranges of excess of 10 mm have been demonstrated in 0.18 μ m technology [162], and it is reasonable that even an interconnect to L3 cache might be possible with minimal or no repeater requirements.

It is important to remember that longer range interconnect are routed in the relatively coarse wiring near the top of the chip, and that adding layers of this scale of wiring is not very expensive. Thus given, the relatively modest amounts of on-chip interconnect anticipated in Figure 6.32 we do not see providing this wiring, at reasonable equalized RC delays, to present any problems within the scope of anticipated technologies.

6.5.2.1.2 Off-chip Wired Interconnect We will discuss two types of off-chip interconnect - extensions of current wired schemes and chip-to-chip schemes assuming some style of 3D interconnect.

Point-to-point electrical links are limited by the frequency-dependent attenuation of traveling waves by skin-effect resistance and dielectric absorption. As these properties of wires do not change as semiconductor technology scales, we do not expect substantial changes in the performance of off-chip electrical links over time.

Today, relatively little attention is given to the power consumed in chip-to-chip communications and thus there is tremendous potential for scaling. For example, a commercial standard 3 Gbps SerDes link consumes approximately 100 mW, and can communicate up to 100 cm in distance. This works out to 30 pJ per bit. There are multiple reasons for this high power consumption. First, the link itself is operating at around 400 mV swings using “always-on” current-mode drivers and receivers. Second, the overhead is significant - for clock and data recovery using a **Phased Locked Loop (PLL)** or **Delay Locked Loop (DLL)** and for the muxes, demuxes, and flip-flops require to interface the slow on-chip data rates to this boosted rate. Overall, today’s commercially

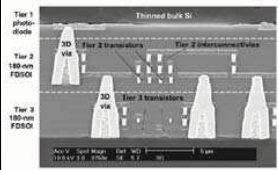
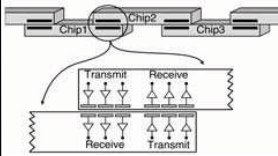
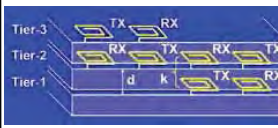
Technology	Pitch	Power	
	Through Silicon Vias	$< 5 \mu\text{m}$	1 -11 fJ/bit
	Capacitive face-to-face	$36 \mu\text{m}$	2 pJ/bit
	Inductive face-up	$30 \mu\text{m}$	0.14 pJ/bit

Figure 6.33: Comparison of 3D chip stacking communications schemes.

deployed links have been designed with little attention to power consumption; the power can be reduced by an order of magnitude or more.

The key to power reduced is to work out how power can be saved while giving a reliable link. A key constraint is the signal to noise ratio (**SNR**) at the receiver. Achieving a **Bit Error Rate (BER)** of 10^{-18} requires a signal to noise ratio of better than 9. (With today's error correction schemes it is generally accepted that power is better spent on increasing signal swing, to reduce error rate, rather than on error correction.) Thus if the noise at the receiver is well controlled, then the driver signal swing can be reduced, even down to 40 mV. Combined with voltage mode circuits and careful management of overhead, a power level as low as 14 mW at 6 Gbps can be demonstrated, or 2 pJ per bit [117]. Note that the bulk of this energy is used, not to transmit or receive the bit, but rather on clock generation and recovery — to generate a transmit clock and recover a receive clock. By 2010 we expect this signaling energy to be reduced to 0.5pJ per bit as more efficient clocking circuits are developed and the entire link is operated from lower supply voltages.

Signaling rates will continue to increase, but as these rates increase, maximum signaling distances will drop. Links operating at 6-10Gb/s are commonplace today. By 2015 we expect that links operating at 30-40Gb/s will be available. These links will operate at reasonable bit error rates up to about 30dB of attenuation. At 30Gb/s this corresponds to about 10m of 24AWG cable or about 1m of PCB stripguide.

6.5.2.1.3 Direct Chip-Chip Interconnect This study anticipates that some form of 3D assembly will be beneficial in an Exascale system. This leads to the possibility of large-scale deployment of direct chip to chip interconnect schemes. A summary of direct chip-chip interconnect schemes, and their state of the art, are summarized in Figure 6.33. **Through-silicon-vias (TSV)** can be used to vertically stack and interconnect wafers and chips. At the time of writing, the state of the art is a 3 wafer stack with less than a 5 μm via pitch. The extra power of communicating

through a vertical via is about that of communicating through the same length of wire. Assuming a $100\mu\text{m}$ via, that works out to 2 - 11 fJ, depending on the signal swing. However, in bulk CMOS the TSV must be passivated, giving an effective dielectric thickness to ground much less than that in the SOI case. Today, the state of the art is a $1\mu\text{m}$ passivation, giving a capacitance for a $100\mu\text{m}$ via of around 44 fJ (equivalent to about $150\mu\text{m}$ of wiring). By 2015 this parasitic should be two to three times better.

By 2015, 3D chip stacking with TSV's will be far enough advanced that the limitations on chip stacks, and via size will be controlled by thermal, power delivery and cost considerations, not basic technology. Sub-micron TSVs have been demonstrated in the laboratory. However, the requirements to get current in and power out will limit practical chip stacks to 4-6 die in any application requiring high performance, high current, logic. 3D technology will be discussed more in Section 6.6.

However, through-silicon via assembly requires extensive integration in one fab facility. A simpler way to vertically integrate chips is to fabricate them separately and communicate through matched capacitors or inductors, forming a series capacitor or transformer respectively. The size must be large enough to get sufficient signal swing at the receiver - roughly $30\mu\text{m}$ is the minimum feasible pitch. The power is higher as the signal swing at the receiver is reduced (100 mV) and a bias network is needed to compensate for the lack of DC information. Power can be reduced by using coding so that no receiver bias network is needed and by using clocked sense-amp style receivers. The power numbers given in Figure 6.33 are all from published sources [107] [69]. There are some limitations to these technologies not often discussed. In particular, the parasitics must be well controlled, and dense grids can not be placed beneath these structures in a metal stack.

6.5.2.2 Switches and Routers

The pin bandwidth of routers for interconnection networks has been increasing at Moore's law rates [81]. We expect this exponential scaling to continue until routers become power limited. The YARC router used in the Cray BlackWidow, for example, has a bidirectional pin bandwidth of 1.2Tb/s . If this router used efficient signaling circuitry with a signaling energy of 2pJ/bit , the power used by the I/O bandwidth of the chip would be just 2.4W . Clearly we are a long way from being power limited in our router designs. By 2015 we expect pin bandwidth to by scale another order of magnitude. In our straw man we assume a modest 7.7Tb/s .

As the pin bandwidth of switches and routers increases, it is more efficient to use this increase by making the radix or degree of the routers higher than simply to make the channels higher bandwidth. In the 2015 time frame, routers with a radix of 128 to 256 channels each with a bandwidth of $60\text{-}120\text{Gb/s}$ will be feasible. Such high-radix routers lead to networks with very low diameter and hence low latency and cost[81]. High-radix networks can use a Clos topology or the more efficient flattened butterfly or dragonfly topologies. The latter two topologies require globally adaptive load balancing to achieve good performance on challenging traffic patterns.

While circuit switching consumes less power than packet switching, it provides less utility as discussed above. However, it is commonly used on the local scale. For example, the Sun Niagara includes a 200 GB/s crossbar in its eight-core 90 nm chip, at a power cost of 3.7 W (6% of a total power of 63 W), or 2.35 pJ/bit . Here the crossbar was used to enable sharing of L2 cache. The energy/bit should scale below 1 pJ/bit by 2015. The energy/bit consumed in a crossbar is dominated by interconnect energy.

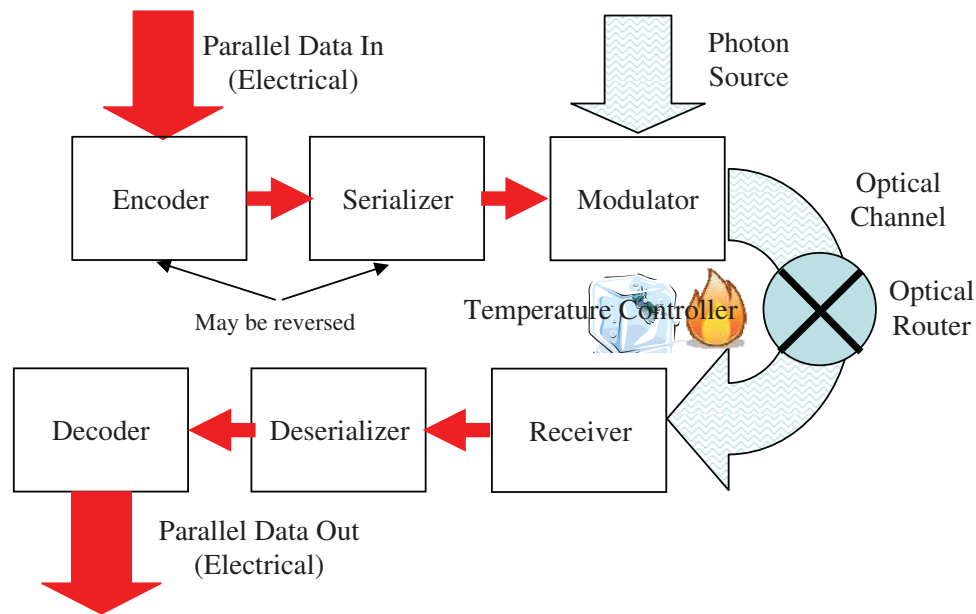


Figure 6.34: Entire optical communication path.

6.5.3 Optical Interconnects

Optical interconnect uses photons instead of electrons for signalling. As such, all communication is point to point, although in recent years optical routing that does not require conversion back to electrical has been introduced experimentally. Both are discussed below.

Also, in these discussions it is important to make fair comparisons with wire-based signalling, especially when considering power. In particular, this means accounting for all possible sources of energy loss in a system, including at least the following (see Figure 6.34):

- The serializers typically needed to go from bit parallel data at the electrical end to very high rate serial electrical bit streams.
- The encoders needed to add whatever error correcting information is needed to permit reliable communication.
- The converters needed to convert from the high rate serial electrical streams to photonic streams.
 - If photons are generated directly from the electrical signal, as with a laser diode, the conversion inefficiency of the process must be included.
 - When modulators are assumed for the conversion process, the power needed for the original photon source needs to be included, and amortized over all channels it sources.
- If some form of optical routing is needed, the power that needs to be expended to generate the routing signal to the router.
- The detection at the receiving end of the photons and conversion back into an electrical form. This usually includes not only the conversion back into electrical form, but also clock recovery.

- The deserialization of the received high speed electrical output from the receiver into a data parallel form (this usually also requires clock recovery).
- Decoders to process the bit stream and correct for errors.
- In addition, many of the emerging electro-optical and optical-optical devices are very temperature sensitive, and heaters or coolers, along with temperature sensors and a feedback controller, may be needed to maintain a constant temperature that permits a stable wavelength.

6.5.3.1 Optical Point to Point Communications

Today, optical links are regularly used for longer range (> 10 m) rack to rack communications. The existence of low loss fiber, means that less energy and volume is used to communicate over these distances using optics, rather than electronics. The open question is to what extent optical communications can displace electrical communications, including at the rack, board, and chip levels? It is commonly agreed that optics offers certain fundamental advantages for interconnect, including low loss, scaling without adding energy, and potential for high wiring density. However, practical issues have always prevented its employment in these shorter range applications. Prime amongst these is power consumption. Electro-optical and optical-electrical conversion has traditionally consumed more power than that gained by the low link loss. Another limiter is the lack of a packet routing technology. While all optical circuit switching is possible, no technology has yet shown real promise of enabling packet routing. Other issues include the lack of highly integrated sub-systems, the relatively low reliability of the III-V devices required (when compared with CMOS), and (often) the need for tight temperature control.

Though there is an active community exploring these issues, and possible directions have been identified, significant R&D investment is required to make short range optical interconnect more useful. The potential for power reduction is being explored, as is the technology for integration. These are being explored in anticipation of bringing optics into mainstream computing. Unfortunately, the current markets for optical components are relatively small, so there is a bit of a “chicken and egg” problem in justifying large-scale commercial investment.

Today, a typical short-range optical link consists of a **vertical-cavity surface-emitting laser (VCSEL)** connected by a multi-mode fiber to an optical receiver. The advantages of this approach include the low-cost VCSEL, and the ease of coupling to a (low-cost) multi-mode plastic fiber. IBM has demonstrated a similar architecture for board-level optical interconnect, replacing the multi-mode fiber with a multi-mode embedded optical waveguide. They have demonstrated this capability at 5 pJ/bit NOT including serial/deserializing multiplexing (SerDes) and **Clock and Data Recovery (CDR)**, at a data rate of 10 Gbps. $250\mu\text{m} \times 350\mu\text{m}$ pitch. Several other groups have demonstrated similar capabilities, at least in part.

However, with the recent emergence of integration into an SOI substrate, a potentially better link architecture is to follow the approach shown in Fig. 6.35. Instead of using a directly modulated laser, a DC laser is used, and a digital modulator employed to provide a signal. A DC laser is more reliable than a directly modulated laser, and less temperature control is needed. Modulation is then done by modulating a fraction of this laser energy. Interference-based Mach-Zender or ring modulators can be used. The receiver can be highly integrated by using a silicon process modified with a Germanium step. As well as improved reliability, this approach offers great potential for improved integration and power consumption. Everything beside the InP laser can be built in silicon. SOI waveguides can be built using a $0.5\mu\text{m} \pm$ wide trace on top of glass. However, to function correctly, the modulators must be connected using single mode waveguides. The modulators rely on subtly

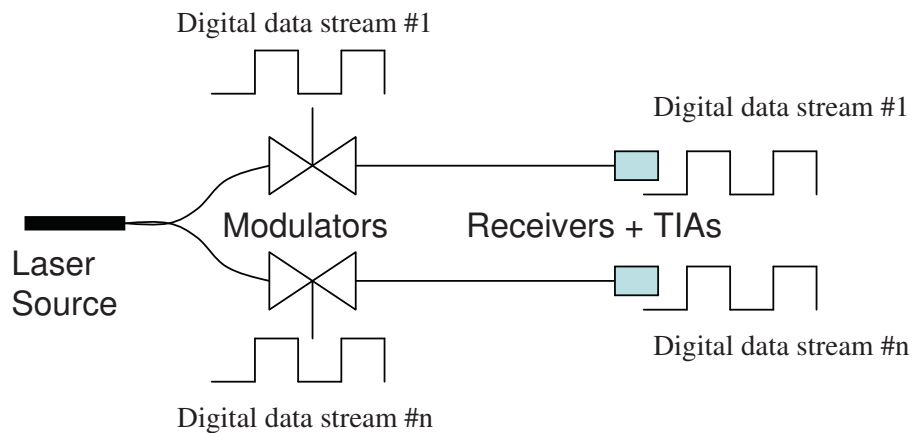


Figure 6.35: Modulator approach to integrated optics.

changing the delay of one light path so as to enable constructive and destructive interference. Thus they only work at one wavelength. The disadvantage of requiring single mode waveguides is that the any external connection, via a single mode fiber, requires sub-micron alignment for acceptable coupling losses (in contrast a multi-mode connection can withstand a multi-micron misalignment). Though pick-and-place machines can be modified to align single fibers, no technology is currently available to align multiple optical waveguides in parallel. Any modified printed circuit board style process would not have the required large-scale registration. Large PCB panels that are available today typically provide $10\mu\text{m}$ registration at best. Sub-micron registration is required to simultaneously align multiple single mode fibers. Step-and repeat lithographic processes provide the required alignment but not over a large area. Nanopositioners could be modified to provide the required precision but this has been little explored.

For example, Luxterra has demonstrated this approach with a 40 Gbps link, connected via single fibers, with a total energetics of 55 pJ/bit, not including SerDes and CDR. However, this speed is not going to be energy efficient, and, at slower rates, it is generally agreed that the energy could be significantly better.

A potential power budget for a near future implementation, based in part on [78] is shown in Table 6.7. The near term column could be considered part of a technology roadmap. Achieving the numbers given in the long term column would require significant investment, and should not be considered as part of the roadmap. Consider the “near term” column first. At 2 Gbps, the receive power needs to be better than 0.2 mW to achieve a BER of better than 10^{-15} . Assuming minimal modulator and connector losses, a source of 0.3 mW is (aggressively) possible. However, this is the DC power consumption. A typical interconnect only sends useful bits for around 10% of the time. This activity factor has to be included in any calculation for energy per bit, based on DC powers, and is accounted for in the Table. A modulator would have an input capacitance of around 100 fF, giving 0.1 pJ/bit at 1 V supply.

The commercial state of the art for optical receivers is 10 mW at 10 Gbps, or 1-10 pJ/bit depending on the assumed activity factor [58]. However, there is potential for scaling. A low capacitance MODFET (modulated-doping field effect transistor) optical receiver has potential for operating at 1 mW total power consumption at 2 Gbps, at acceptable BERs, giving the receive energies listed in Table 6.7. Again, these amplifiers consume DC power. The total link power of a

Component	Near Term	Long Term
Laser @ 2 Gbps 10% activity factor (AF)	0.3 mW per channel 0.15 pJ/bit 1.5 pJ/bit	same? same
Modulator	0.1 pJ/bit	0.01 pJ/bit
RX + TIA 10% activity factor	0.5 pJ/bit 5 pJ/bit	0.05 pJ/bit? 0.05 pJ/bit?
Sub-total 100% AF 10% AF	0.75 pJ/bit 7.5 pJ/bit	0.21 pJ/bit 1.5 pJ/bit
Temperature Control	?	?

Table 6.7: Energy budget for optical modulator.

short-term scaled link is thus 0.75 pJ/bit for 100% activity factor and 7.5 pJ/bit for a 10% activity factor. While the power achieved is not low enough to replace chip to chip links at the board and backplane level, this level of power consumption has potential to favor optical interconnect at distances over 50 - 100 cm.

The right hand column in Table 6.7 assumes aggressive long term scaling of the key technologies. These technologies should not really be considered part of the “roadmap” as considerable R&D investment would be required to achieve the listed potential. Quantum Well Modulators have potential to scale to 10 fF input capacitance, or smaller, permitting a ten-fold reduction in modulator power consumption [30]. New classes of optical MOSFETs have the potential to allow optical receivers to operate in voltage, rather than current, mode and reduce the power down to levels comparable with a large CMOS gate [114]. These have potential to operate at power levels of 0.05 pJ/bit and even lower.

One power consumption that is not included in Table 6.7 is that required for cooling. Optical modulators require tight temperature control for correct operation, typically to $\pm 10^\circ\text{C}$. Typically, active temperature control is used, often **Thermal Electric Coolers (TEC)**. The additional power that would be consumed operating an active cooler will depend on the total heat load at the spot being cooled. Thus it would be a required overhead but difficult to estimate at this stage.

6.5.3.2 Optical Routed Communications

By adding additional modulators, the approach outlined in Figure 6.35 can be extended to build an all-optical circuit switched architecture. Since each additional modulator consumes only 0.1 pJ/bit, the additional power over that shown in Table 6.7, would be minimal. Of course the (probably) electrical network that must route the switching commands to the modulators must also be taken into account. Thus, if modulator based optical interconnects were employed in an Exascale computer, interesting circuit-switched functions might be added for minimal extra power and cost. For example, Bergman proposes a low-latency crossbar architecture for future multi-core computers[127].

However, optics still lacks the capability to enable the switching function most useful to the interconnect network of an Exascale computer - a packet router. Packet routing requires that routing header information accompany the data, so that the path taken by the packet can be set up and taken down as the packet passes through the switch. This reflects the usually unpredictable nature of intra-computer communications. To date, optical switches require a separate electrical network for circuit set-up and re-route. This introduces an overhead of 10s' of ns every time the circuit is reconfigured. Thus they are best suited for computational tasks that benefit from node-

to-node bandwidth that changes substantially with each task, or for tasks that benefit from very large bursts of data being routed between CPUs. At this stage, there is little evidence that the potential throughout benefit of such switching would justify reducing another resource in order to account for the required power, except for those links where optics is already justified over electrical interconnect, i.e. longer inter-node links.

6.5.4 Other Interconnect

Other possible interconnect technologies that could impact HPC in the future include the following:

- **Carbon Nanotubes (CNT).** Due to their excellent conductivity, carbon nanotubes have potential to permit an increase in wire density while also improving latency and power consumption [59]. They can be routed tightly at low resistance, with a pitch better than $0.8\mu\text{m}$. Cho et.al. [59] predict a power consumption of 50 fJ/bit/mm at full swing. Thus a power consumption of 6 fJ/bit/mm at reduced swing would be reasonable. However, given the currently limited state of demonstration and the open technical issues involving patterning, contacts, etc., it is unlikely that CNTs will be mature enough to be part of a 2105 Exascale computer chip.
- **Nano-enabled Programmable Crosspoints.** The emerging resistive memories discussed earlier in Section 6.3.5 also have potential to serve as the base technology for high-density switch-boxes. These could reduce the area and power impact of SRAM-based programmable switch boxes by almost an order of magnitude, enabling new ideas in configurable computing to be investigated. So far, this concept has been mainly explored in specific applications, so its potential impact on HPC is largely unknown. At the least, it would outperform the crossbar switches discussed above, in at least power per operation. Given that a roadmap exists for such memories to be commercially deployed next decade, their incorporation into logic CMOS devices is plausible and should be considered.

6.5.5 Implications

The summary roadmap for interconnect is presented in Table 6.8. The three metrics evaluated are wire density, power/bit and technology readiness. The energy/bit numbers come from the discussion above. Wire density is for a single layer of routing only. On-chip long-distance wires are assumed to have $2\mu\text{m}$ width and space. Chip-to-chip routing is assumed to have 1 mil ($25\mu\text{m}$) width and 4 mil space (to control crosstalk). Note these numbers are per-wire. Differential routing halves these numbers. PCB-embedded multi-mode wires have been demonstrated at $100\mu\text{m}$ width and $1000\mu\text{m}$ pitch. Single-mode waveguides can be much narrower but (as yet) can not be fabricated on a large panels.

Overall, it is fairly clear that copper interconnect is here to stay, at least for the time frame of an Exascale computer. Today the cross-over point between optics and electronics, in terms of energy/bit and \$/Gbps is at long distances - several meters. However, optical interconnect is highly unoptimized. Unfortunately, this is largely due to the lack of a volume market for optimized optical interconnect. With new, better optimized devices and circuits, together with greater integration, the cross-over point is likely to shrink to something better than 100 cm, perhaps 50, perhaps shorter.

One significant advantage of optics is that, if desired, circuit switching can be added to point-to-point optical interconnect with relatively little overhead. However, these new technologies assume single-mode fiber, requiring sub-micron alignment. While the technology for single fiber alignment

Technology	Density (wires/mm)	Power (pJ/bit)	Technology Readiness
Long-range on-chip copper	250	18 fJ/bit-mm	Demonstrated
Chip-to-chip copper	8	2 pJ/bit. Includes CDR	Demonstrated. Potential for scaling to 1 pJ/bit
Routed interconnect in 2015	n/a	2 pJ/bit router or non-blocking circuit switch 1 pJ/bit	roughly the same for packet
Optical State of Art (multi-mode)	10	9 pJ/bit. NOT including CDR	Demonstrated.
Optical (Single mode) in 2010	300	7.5 pJ/bit SOI waveguides PCB-embedded waveguide does not exist	Assumes lithographed
Optical (Single mode) in 2015	300	1.5 pJ/bit	At early research stage
Optical Routing		Add 0.1 pJ/bit (2010) for each switch	
Optical - temperature control			TEC cooler demonstrated
CNT bundles	1250	6 fJ/bit-mm	Undemonstrated

Table 6.8: Summary interconnect technology roadmap.








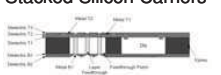
Approach	Wires/mm or sq.cm  e.g. 2 wires/mm 1 mm	Bandwidth/mm Routable signal pairs per mm per layer * # layers * bit-rate per signal pair	Comments
Laminate (Ball Grid Array) 	20 wires/mm/layer (2-4 signal layers) ~2,000 max total pin count	6 pairs/mm @ 30 Gbps = 180 – 720 Gbps/mm (1-4 signal layers) Package I/O: 500 pairs = 15 Tbps	1 mil line/trace presents practical limit. 1 mm BGA ball pitch
Silicon Carrier 	50 wires/mm/layer (2 signal layers) Has to be packaged for I/O	12 pair/mm @ 30 Gbps = 360 – 720 Gbps/mm (1-2 signal layers)	2 signal layers is practical limit.
3DIC Stack 	~10-40 wires/mm vertically around edge	Total: 100 – 200 pair @ 10 Gbps → 0.5 - 2 Tbps (assumes memory)	Limited interconnect performance
3D IC with Through Silicon Vias 	In excess of 10,000 vias per sq.mm.	In excess of 100,000 Tbps/sq.cm. Really determined by floorplan issues	Chip stack limited to 4-8 chips, depending on thermal and other issues
Stacked Packages 	1/mm on periphery	25 pairs total @ 10 Gbps → 250 Gbps	Not very applicable to high performance systems
Stacked Silicon Carriers 	Vertical connections @ 20 um pitch → 250,000 / sq.cm	62500 pairs @ 30 Gbps → 1900 Tbps / sq.cm	Limited by thermal and coplanarity issues.
Stacked Silicon Carriers 	Vertical connections @ 100 um pitch → 10,000 / sq.cm	2500 pairs @ 30 Gbps → 75 Tbps / sq.cm	Early demonstration only. Air cooled to < 117 W total.

Figure 6.36: Representative current and future high-end level 1 packaging.

is mature, there is no equivalent of the Printed Circuit Board in large scale single mode interconnect. A significant investment would be needed for this technology to be available.

6.6 Packaging and Cooling

6.6.1 Packaging

For convenience, packaging is generally considered within the scope of a hierarchy, as follows:

- **Level 1 Packaging** connects, contains, and cools one or more silicon die and passives, such as capacitors and resistors. A typical level 1 package would be an organic (plastic) surface mount.
- **Level 2 Packaging** connects a set of level 1 packaged components on a common substrate, such as a printed circuit board (PCB).
- **Level 3 Packaging** interconnects a set of level 2 packages. A typical structure would be a backplane or midplane, into which a number of PCBs are plugged using separable connectors. Commonly called a “rack,” a level 3 package might span a meter or more in distance and connect several tens of PCBs.

In an Exascale computer, Level 3 packaging and rack to rack interconnectivity is not trivial, but is not driven as much by technology as the other two, and is thus not discussed further. The one key issue of connectivity is typically provided today by cables, electrical for shorter ranges and optical for longer ranges.

6.6.1.1 Level 1 Packaging

An extrapolation of future capabilities based on current level 1 packaging technologies is given in Figure 6.36. The most ubiquitous package technology today is based on organic (i.e. plastic) **Ball Grid Arrays (BGA)** onto which chips are flip-bumped. The leading edge state of the art is typified by the Endicott Interconnect Technology HyperBGA product [71]. This BGA supports 1 mil features and uses laser drilled vias to achieve high densities. Nonetheless, it is limited to a maximum pin-out of around 2,000 pins, on a 0.5 - 1 mm grid. It represents the current limit for manufacturing for laminate technologies. Exceeding these limits would require a level of precision beyond that of today's equipment, i.e. would require a sizeable R&D investment. Assuming half the pins are used for power distribution, a 2,000 pin package can only support 500 signal pairs. Assuming a future 30 Gbps capability on these pairs, that amounts to 15 Tbps, or 2 TBps total in and out of the package. A previous section suggested a CPU I/O requirement of 0.7 - 10 TBps. A HyperBGA package could support the low-end, but not the high-end of this range. Even at the low-end, breakout to the next level of packaging with good noise control (see below) might be difficult.

One way to increase the wire density beyond current laminate technologies would be to use lithography instead of screen-printing. Lithography can be used on a silicon carrier to create very high wire densities, down to $10\mu\text{m}$ line and space, though coarser features are often used so as to improve yield. However, most current silicon carrier technologies are limited to 4 layers - power, ground and 2 signal. Thus, the total wire density per unit of cross-section, for the entire wire stack, ends up being about the same as that for a high-end laminate, due to the latter's higher signal layer count! This is due to planarity limitations introduced by the requirements for thick dielectrics so that transmission line structures can be built. New concepts would be needed to make high layer count silicon carriers. Note that the silicon carrier itself must be packaged, for example, on top of a laminate.

The next row in Figure 6.36 shows commercial state-of-the-art 3D chip stacks, built using wire-bonds or some form of stacking and edge processing [1]. These are often used for memories, so as to improve their form factor and physical density. However, their I/O is relatively limited and won't support a stack of high bandwidth DRAMs as anticipated in this study. (Note, 10 Gbps I/O rate is used here, rather than the more aggressive 30 Gbps to reflect the reduced capability of logic built in a DRAM process.)

Through-Silicon Vias (TSV) can enable a 3D chip stack with very high internal connectivity. This is a technology that is likely to be mature by 2015, partly because of current DARPA funded efforts. The available vertical bandwidth is very high, and is limited by practical considerations such as silicon area tradeoffs, etc. Several vendors are designing 3D integratable memories. However, it is difficult to envision a vertical chip stack consisting of more than four to eight chips. There are many practical reasons for this. First current has to be brought in and heat removed. Doing either through a large number of chips is highly impractical except for very low-power circuits. Improving power delivery or heat removal capability requires that more silicon vias be added to all layers, as more chips are stacked. Second, test and yield issues works against high layer counts. Each chip in the stack must either be pretested before integration, or methods to cope with the accumulated yield loss introduced. Unfortunately, even an eight-chip stack (which is unlikely to have acceptable

heat removal) does not integrate sufficient memories with a CPU to meet the anticipated memory needs.

Two other currently emerging technologies use silicon carriers to create 3D packages. One approach involves adding additional interconnect layers and chips to a single carrier. Yet to be demonstrated, there would be concerns about maintaining planarity and cooling. Since the chips are essentially encased in polymer, heat removal would be as difficult as for the 3D case.

Another approach, currently being demonstrated by Irvine Sensors, involve stacking separate silicon carriers using interconnect studs. Stress issues limit the width of the structure to two to three cm. However, one significant advantage of this approach over the previous discussed one is the ability to include heat spreaders in the 3D package. Irvine plans to demonstrate an air cooled package, capable of dissipating a total of 117 W. With additional innovation, greater thermal capacities would be possible.

6.6.1.2 Level 2 Packaging

Level 2 packaging almost invariably consists of a **Printed Circuit Board (PCB)**. The state of the art supports high capacity in one of two ways. The first way is to use fine line technologies (1 mil traces) and laser drilled vias. However, the layer count is limited in this approach. The second is to go to high layer counts and use conventional line widths: e.g. use a 0.2" thick PCB, and support 4 mil (100 μ m) wide traces. Such a board might support 10 X-direction routed signal layers, 10 Y-direction routed, and 20 power/ground layers. Using normal routing rules for differential pairs would give a density of one pair every 28 mil (crosstalk forces the pairs apart). The maximum cross-section bandwidth would be 28 pairs per signal layer per cm, or 280 pairs per cm (10 layers), or 2800 Gbps (at 10 Gbps per pair for memory). This translates into a potential cross-section bandwidth of 350 GBps per cm at 10 Gbps or 1.05 TBps at 30 Gbps.

A combination of a high end **Single Chip Package (SCP)** with a high end, thick, PCB would enable such a packaged CPU to have a total bandwidth of 2 TBps with 30 Gbps signaling, and 666 GBps with 10 Gbps signaling. The limit is set by the package technology, not the board technology. Today, there are no investments going on to improve the capacity of conventional packaging. This would work if the required memory bandwidth was at the low end of the scale (16 memories \times 44 GBps = 704 GBps) but not if larger memories or memory bandwidths were required.

Moving to a silicon carrier, single tier or stacked, does not immediately alleviate the situation. A 1 sq. cm die could provide a total peripheral off-chip bandwidth of 600 GBps at 10 Gbps and 1.8 TBps at 30 Gbps. However, with careful yield management (using finer lines for short distances) and planning, the I/O density could be increased locally just at the chip edge, to support possibly two to three times this bandwidth. An example is given in [103]. However, this solution would still not support the higher end of the possible bandwidth and capacity requirements.

Possible 3D solutions to providing sufficient memory bandwidth and memory capacity are discussed in Chapter 7.

6.6.2 Cooling

An Exascale computer presents a number of novel thermal design challenges, including the following:

- At the module level, any potential 3D chip assembly must be cooled sufficiently to ensure reliable operation, to limit leakage power, ensure timing budgets are predictably met, and to guarantee DRAM refresh times are accurate. Heat fluxes of up to 200 W/sq.cm. are anticipated.

Approach	Thermal Performance	Comments
Copper Heat Spreader	Thermal conductivity = 400 W/(m.K)	
Diamond	Thermal conductivity = 1000 - 2000 W/(m.K)	Expensive
Heat Pipe	Effective conductivity = 1400 W/(m.K)	Very effective
Thermal Grease	Thermal conductivity = 0.7 - 3 W/(m.K)	
Thermal vias with 10% fill factor	Effective Conductivity = 17 W/(m.K)	
Thermal Electric Coolers	Limited to less than 10 W/cm ² and Consumes Power	
Carbon Nanotubes	Excellent	Early work only

Table 6.9: Internal heat removal approaches.

- At the rack level, the total thermal load is likely to be in the range of 10-200 KW. A solution is required that supports this level of cooling while maintaining required temperatures and permitting provisioning of high levels of system interconnect.
- At the system level, the total thermal load of an Exascale computer is anticipated to be 10s of MWs, and this heat load must be properly managed.

6.6.2.1 Module Level Cooling

The main objective of module level cooling is to control the chip junction temperature in order to minimize potential failure mechanisms such as electro-migration, and to minimize leakage currents in memories and logic transistors. Typical objectives are in the 85 C to 100 C range. Generally, chip level cooling is evaluated in terms of a thermal resistance:

$$R_{\theta} = \Delta T / Q \quad (6.10)$$

where ΔT is the temperature drop and Q the total heat being transferred. For convenience, thermal resistance is typically broken into two components, internal and external resistance. The internal resistance is determined by the conductivity of the materials and structures between the chip junction and the circulating coolant. The external resistance is determined by the ability of the circulating coolant to remove this heat away from the chip vicinity.

Some of the available structures that are used for the internal portion of the heat flow are summarized in Table 6.9. The last entry in this table is for an array of tungsten thermal vias that might be used in a 3D chip stack. Assuming that 10% of the chip area is given over to thermal vias, then the effective thermal conductivity is 17 W/(m.K). This calculation is only included to illustrate the relative difficulty of cooling chips internal to a 3DIC. **Thermal Electric Cooling (TEC)** is included as a reference. TECs can remove heat without any temperature drop. However, the supportable heat flux is limited and they consume almost as much electrical power as they conduct heat power. Thus they are unlikely to be used in an Exascale computer except for points requiring critical temperature control, such as optoelectronic modulators.

Available external cooling mechanisms are summarized in Table 6.10. The most common cooling mechanism is forced air cooling using fans and heat sinks. There have been several demonstrations

Approach	Thermal Performance	Comments
Air (Finned Heat Sink)	$R = 0.6 - 1.0K/W$	Individual Heat Sink & Fan for 12 mm die Can dissipate up to 100 W for 60 K rise
Water (Channel Heat Sink)	$R = 0.3 - 0.6K/W$	Individual Heat Sink & Fan for 12 mm die Can dissipate up to 170 W for 60 K rise Requires 0.1 bar pump
Immersion	$R = 0.4K/W$	
Microchannel and Cooling capacity of $300 - 800W/cm^2$	pump 0.3 - 8 bar	
Two-phase	$R = 0.1K/W$	
Spray Cooling and Cooling capacity of $300 + W/cm^2$		
Refrigeration and $R = 0.05K/W$	Consumes more power than heat removed	

Table 6.10: External cooling mechanisms.

in which 100 W have been air cooled satisfactorily [161]. Air cooling capacity can be increased a little by using larger heatsinks and louder fans.

Water cooling using a microchannel cooling plate built as part of the package can handle up to 170 W based on existing technology [161]. The water is pumped through the cooler and then circulated to an external air cooler. With cooling channels of around 1 mm width, a pump pressure of around 0.1 bar is needed. Narrower channels improve thermal performance at the cost of bigger pumps. Another concern with water cooling are the prevention and management of leaks.

Direct immersion in a non-water coolant, such as a FC-72, can be used to simplify the overall cooling design. However, since these coolants do not offer the thermal performance of water, the overall cooling solution tends to provide the same as direct water cooling. For example, [82] recently demonstrated a cooling performance equivalent to 0.4 K/W.

Other techniques have been long investigated but have never been implemented. Reducing the channel size and integrating the channels into the silicon chip has been demonstrated many times, and can improve thermal performance further. However, these introduce further mechanical complexity, increase the potential for leaks and require stronger and more power hungry pumps. Two-phase cooling (i.e. boiling) improves heat removal capacity (e.g. [74]) but at the expense of complexity and the need to carefully manage the boiling interface. In this work, using $100\mu m$ channels, 2W of pump power was needed to dissipate 3W from the chip, illustrating the power cost of the pumps required for microchannel cooling. Spray cooling can support heat fluxes in excess of $300W/cm^2$ with a better than 60 C temperature drop, but, again, at the expense of complexity. Through refrigeration can be very effective it is unlikely to be used here due to the added power consumption it implies. Cooling to liquid nitrogen temperatures takes twice as much work as the heat being removed. Even cooling from an ambient of (say) 40°C to 10°C, takes 11% more work than the heat being rejected.

Another issue that often rises in cooling design is the complexity introduced by the requirement to get the same power, electrically, that is being extracted thermally. With a DC limitation of

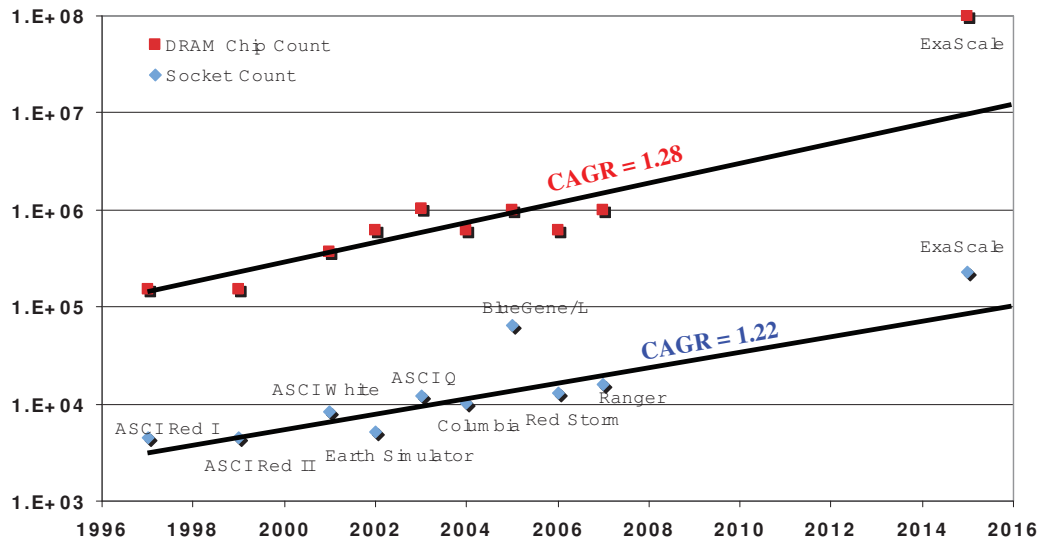


Figure 6.37: Estimated chip counts in recent HPC systems.

about 1 A per solder bump, 200 bumps are required just to distribute DC to CPU chip. Since bumps will have to be added to reduce power/ground inductance so as to control simultaneous switching noise, over 1,000 bumps might be needed unless alternative power delivery technologies arise. Unfortunately, the bumped side of the chip is not terribly effective at removing heat.

A typical solution is to use the front side of the chip to get DC power in, and the backside to get heat out. This approach is almost universally used. However, it complicates potential 3D designs. This will be discussed further in Chapter 7.

6.6.2.2 Cooling at Higher Levels

No matter whether air or water cooling is used at the module level, it is very likely that the machine as a whole will rely on air cooling above the module level, particularly as we go into either the departmental and especially the data center class Exascale systems. This will require careful design of the plenums etc., so as to efficiently manage air flow without excess noise.

For a data center system as a whole, at very best the target 20 MW of power has to be dissipated, even if simply vented externally. If the external ambient is higher than desired, then air conditioned cooling is required at least on the incoming air. To bound the problem, assume that 20 MW of cooling is required. That is equivalent to around 70 M BTU/hour. An air conditioner with a (future) SEER rating of 20 would consume 350 KW to provide this amount of cooling. Overall up to 5% of the power budget is consumed in system cooling.

6.7 System Resiliency

While the failure rate of any particular component may be relatively small, the resiliency of a computing system depends strongly on the number of components that it contains. This is particularly true of the data center class systems, and as such is the focus of this section.

	Hardware	Software	Network	Environment	Human	Unknown
% Breakdowns	62%	18%	2%	1%	1%	16%
% Downtime	60%	19%	1%	2%	0%	18%

Table 6.11: Root causes of failures in Terascale systems.

As shown in Figure 6.37 the number of components in recent supercomputing systems is increasing exponentially, the compound annual growth rate of memory (at 1.28X per year) exceeding slightly that of sockets (at 1.22X per year). Assuming continuation of past system growth, a 2015 system will consist of more than 100,000 processor chips (“sockets”) and more than 10 million DRAM chips. While an Exascale system may require more aggressive system scaling, even past scaling trends will present substantial resiliency challenges.

6.7.1 Resiliency in Large Scale Systems

The supercomputing community has gained experience with resiliency through Terascale and emerging Petascale machines. Terascale systems were typically designed from standard commercial components without particular resiliency features beyond replication of components such as power supplies. Schroeder and Gibson [124] and Gibson[50] analyzed failure logs for these machines and reported that Terascale systems with thousands of sockets experienced failures every 8–12 hours, corresponding to 125–83 million FIT (failures in time, which is failures per 10^9 hours). The study also showed that the greatest indicator of failure rate was socket count; across all of the systems they examined, the average failure rate was 0.1–0.5 fails per year per socket (11–57 KFIT per socket). Schroeder and Gibson also analyzed the root causes of failures, with the averages summarized in Table 6.11. Their results show that hardware is the most dominant source of failures, including both intermittent and hard failures.

More recent systems, such as IBM’s BlueGene Supercomputer, a 64K socket system with a Top500 performance of 280 TFlops, achieve better resiliency than reported by Schroeder [5]. The BlueGene chips and systems were designed with a resiliency (FIT) budget that is much more aggressive than the capabilities of Terascale systems. The FIT budget summarized in Table 6.12 shows that power supplies are most prone to failure and that while expected failures per DRAM chip is small, the sheer number of chips make DRAM the largest contributing factor to failures. Nonetheless, the overall FIT budget for the entire system is only 5 million (76 FIT per socket or 0.001 failures per year per socket), corresponding to a hardware failure rate of once every 7.9 days. Assuming that hardware accounts for only half of the failures, the aggregate mean time to interrupt (MTTI) is 3.9 days. The source of improved failure rates stems from more robust packaging and enhanced hardware error detection and correction.

Additional studies have measured failure rates due to specific causes in the system. Schroeder and Gibson examined disk drive reliability and report that disk drives are the most frequently replaced components in large scale systems [125]. However, disk drives have not traditionally dominated the cause of node outages because they are often replaced pro-actively when they begin to show early warning signs of failure.

In a different study, Michalak et al. examined the susceptibility of high performance systems to uncorrectable soft errors by monitoring a particular memory structure protected only by parity [104]. They report one uncorrectable single event upset every 6 hours (167 million FIT), reinforcing the need to design for **Single Event Upset (SEU)** tolerance.

Figure 6.38 shows the expected error rates as a function of socket count and three different per-socket failure rates ranging from 0.1 (representing the best observed failure rate from Schroeder)

Component	FIT per Component	Components per 64K System	FIT per System
DRAM	5	608,256	3,041K
Compute + I/O ASIC	20	66,560	1,331K
ETH Complex	160	3,024	484K
Non-redundant power supply	500	384	384K
Link ASIC	25	3,072	77K
Clock chip	6.5	1,200	8K
Total FITs			5,315K

Table 6.12: BlueGene FIT budget.

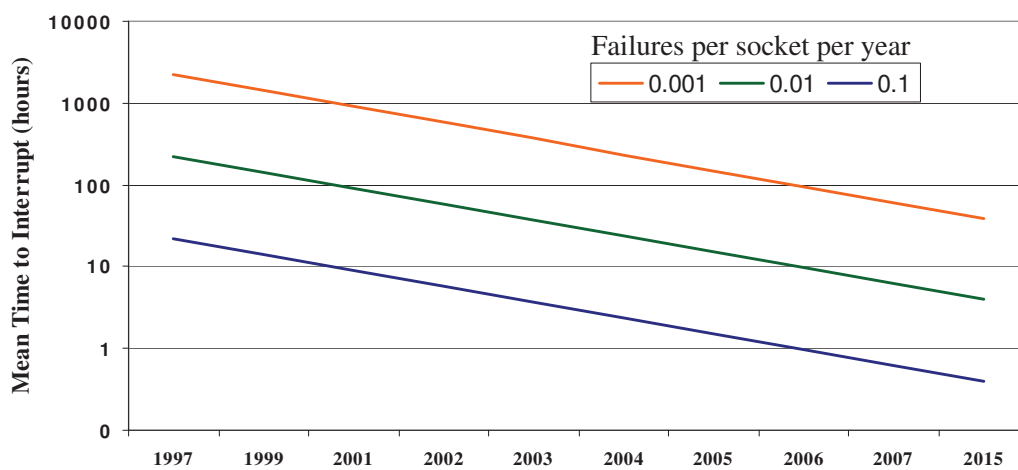


Figure 6.38: Scaling trends for environmental factors that affect resiliency.

to 0.001 (representing a system with aggressive resiliency). The number of sockets is assumed to increase at 25%/year to match a performance demand of 2x system performance per year and 2x socket performance every 18 months, reaching 220K sockets in 2015. Schroeder's per-socket resiliency assumptions results in a MTTI of 24 minutes, while a factor of 10 improvement in resiliency results in a failure every 4 hours.

6.7.2 Device Resiliency Scaling

While the above analysis assumed constant device failure rates, in fact technology scaling will make sustaining chip-level reliability substantially more difficult, with three major causes:

- **Hard Failures:** Shrinking feature size is a root cause of increased susceptibility to hard failures and device wearout. The ITRS identifies more than 20 difficult short-term semiconductor reliability challenges, including dielectric breakdown, thermal stresses, and electro-migration[13]. Because device dimensions will continue to shrink while the power supply voltage will level out, electric fields will increase and contribute to several different breakdown modes. Temperature is also a substantial contributor to device failure rates, increasing the importance of thermal management in future systems. The ITRS sets as a goal 10-100 FITs per chip in the coming generations but recognizes that there are no known solutions for 32nm and beyond.
- **Single-Event Upsets:** Single event upsets (SEU) are influenced primarily by node capacitance in integrated circuits. DRAM cells are becoming less susceptible to SEUs as bit size is shrinking, providing a smaller profile to impinging charged particles, while node capacitance is nearly constant across technology generations. Logic devices, latches, and SRAMs are all becoming more susceptible due to the drop in node capacitance need to scale to higher circuit speeds and lower power [128]. Researchers predict an 8% increase in SEU rate per bit in each technology generation [63]. The ITRS roadmap sets as a goal a steady 1000 FIT per chip due to SEUs and indicates that potential solutions may exist [13]. However, enhanced microarchitectural techniques will be required to mask a large fraction of the chip-level FITs and achieve a satisfactory level of resiliency.
- **Variability:** As transistors and wires shrink, the spatial and temporal variation of their electrical characteristics will increase, leading to an increase in speed-related intermittent or permanent faults in which a critical path unexpectedly fails to meet timing. Researchers predict that the threshold voltage for transistors on the same die could easily vary by 30% [19]. Managing variability will be a major challenge for process, device, circuit and system designers.

Figure 6.39 summarizes several of these effects as a function of time, with a projection into the Exascale time frame.

6.7.3 Resiliency Techniques

As resiliency has become more important in the high-performance commercial marketplace, microprocessor designers have invested more heavily into means of detecting and correcting errors in hardware. These techniques typically fall into the following categories:

- **Encoding:** Parity and SECDED codes are commonly used to detect or correct errors in memory structures and in busses. Encoding provides low-overhead protection and on-line

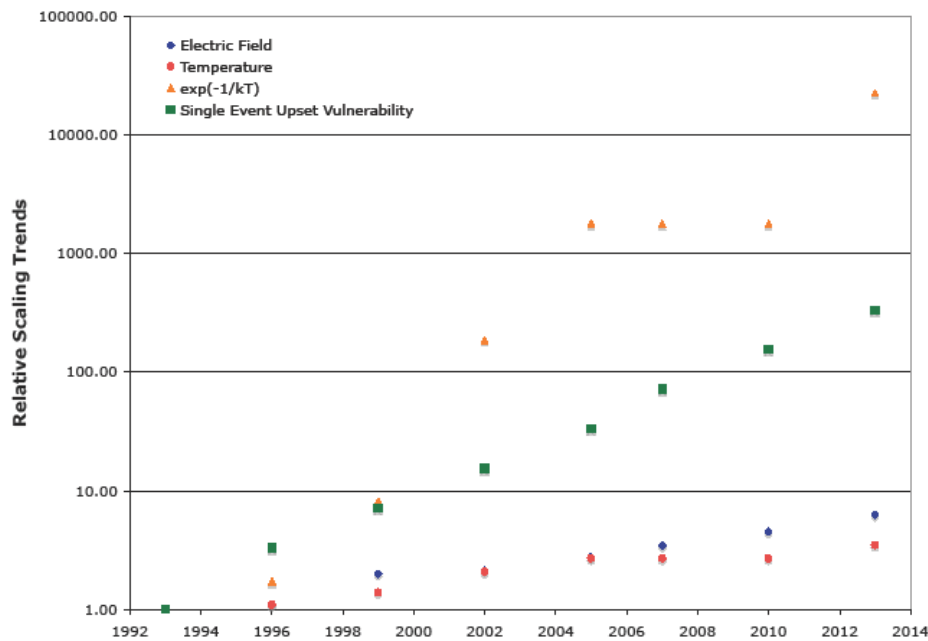


Figure 6.39: Increase in vulnerability as a function of per-socket failure rates.

correction of corrupted bits and can be used to protect against intermittent single upset events as some forms of permanent hard failures. Encoding overheads are typically in the 5-10% range for those structures to which encoding can be applied.

- **Scrubbing:** Memory scrubbing involves frequently reading, correcting, and immediately rewriting regions of memory, and eliminates latent errors in stored data before it is used by flushing unused data items from the storage and fixing any correctable data elements. Scrubbing can be applied to memory structures, caches, and register files and is typically a means of reducing the likelihood that a single-event upset will cause a program failure. Scrubbing typically incurs very little area and time overhead.
- **Property Checking:** Many operations in a processor or system can be verified by checking properties during execution. While a simple example is a bus protocol in which no two clients should be granted the bus at the same time, this technique can be widely applied to different parts of a processor and system. Property checking can be implemented with relatively little overhead.
- **Sparing:** Hard component failures can be tolerated, often without system failure, by providing a spare component that can be swapped in. This technique can be applied at the chip level (spare processing elements or rows/columns in a DRAM) or at the system level in the form of spare power supplies or disk drives. Sparing relies upon some other mechanism for detecting errors. Sparing is typically considered as 1 of N in which a spare can be swapped in for one of several components. A larger value of N results in less overhead.
- **Replication:** Replication can be used for both detection and correction. Detection typi-

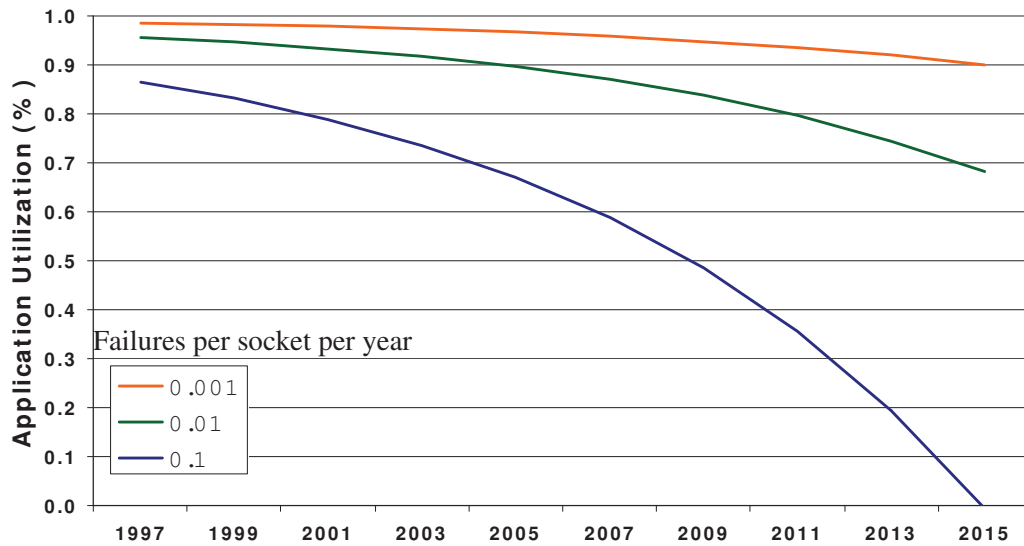


Figure 6.40: Projected application utilization when accounting for checkpoint overheads.

cally requires two parallel versions of the component to operate simultaneously and compare outputs. An error is detected when the comparisons mismatch, and requires some alternate form of recovery. **Triple modular redundancy (TMR)** provides three copies of the component and votes among the three with the majority providing the defined correct output. Replication is much more expensive, requiring 100-200% overhead.

The IBM Power6 microprocessor incorporates encoding, scrubbing, property checking, and sparing as a means to increase resiliency and reduce the FIT rate [120]. Processors such as the IBM G5 replicate significant pieces of the pipeline in a self checking mode [138]. Highly available systems, such as the Compaq Nonstop Himalaya, employed TMR to minimize the down-time in mission critical systems. Modern supercomputers supply fail-over spare power supplies within a chassis (tens of nodes) as well as spare power circuits in the power distribution network.

6.7.4 Checkpoint/Rollback

At the system level, recovery is often implemented using a checkpoint-rollback scheme, where a **checkpoint** is the copying of enough of application memory to an alternative storage medium in a fashion that allows for the application to be stopped and then at some arbitrary time to be restarted by moving the copied data back from this medium (**rollback**). Today, most such schemes are “application-agnostic,” that is do not try to minimize the amount of data copied by selection. Thus, for sizing purposes, a checkpoint operation requires copying essentially all of data memory.

Typically, the application’s execution must be suspended during this checkpointing period so that the data being checkpointed is consistent. This dead time represents an overhead that reduces system utilization, and thus effective system application-level performance.

The cost of a checkpointing scheme depends on the time to take a checkpoint, the checkpointing interval, time to recover, and the rate at which recoveries are necessary (typically correlated with the MTTI). As an example, BlueGene/L aims for a single-checkpoint cost of 12 minutes for application-

initiated checkpointing rollback/recovery, and employs a several techniques to reduce the overhead, including incremental checkpointing and memory hashing [115]. At this cost, a checkpointing interval of 2.5 hours will result in an overall 8% overhead when faults do not occur (8 minutes/150 minutes), meaning that the system can execute user applications at most 92% of the time (if no faults occur requiring rollback).

Knowing the time to perform the checkpoint (t), the period between checkpoints (p), and the mean time between interrupts (MTTI) allows this utilization to be computed as[50]:

$$(1 - AppUtilization) = (t/p) + p/(2 * MTTI) \quad (6.11)$$

with a minimal overhead found when:

$$p = \sqrt{2 * t * MTTI} \quad (6.12)$$

This actual utilization percentage must then be applied to the performance of the system as a whole to come up with a “sustained performance.” Figure 6.40 shows the projected application utilization rates of large scale systems as a function of machine size and MTTI resulting from 0.1–0.001 failures per socket per year. The time to take a checkpoint is held constant at 12 minutes, and the checkpointing is assumed to occur at optimal intervals. Rollback/recovery overhead quickly dominates the application, rendering the machine useless using observed Terascale failure rates.

Extrapolating this to Exascale systems, if the checkpoint time is similar, but the equivalent failure rate per socket does not improve from its current 0.1 value, then MTTI will remain in the few hours time scale. Also, if we assume that Exascale memory will be orders of magnitude greater than the Terascale machines used for this study, it is highly likely that, unless extraordinary increases in bandwidth the checkpointing memory is made, that this 12 minute checkpoint time must itself escalate to the point where the machine is useless. Figure 6.30 in Section 6.4.1.3 gives an absolute minimum time per petabyte of about 15 seconds for a system in 2015 with 1 Exabyte in about 150,000 drives running at 100% of max bandwidth. For a 10 PB disk system running at perhaps 30% of peak bandwidth and supporting the low-memory 3.6PB configuration of the aggressive strawman of Section 7.3, this time might balloon out to nearly 5 hours - clearly a ridiculous number! Thus, in the absence of significantly new storage technologies, per socket and machine MTTI **must** be driven down in order for applications to make good use of large scale machines.

6.8 Evolution of Operating Environments

Operating environments represent the collection of system software that manage the totality of computing resources and apply them to the stream of user tasks. As described in section 4.2, conventional operating environments comprise node operating systems, core compilers, programming languages and libraries, and middleware for system level resource allocation and scheduling. It also supports external interfaces to wide area networks and persistent mass storage but these are dealt with elsewhere in this report.

Near term evolution of operating environments is required to address the challenges confronting high performance computing due to rapid technology trends described earlier in this report. Among these are the reliance on multi-core processor components to sustain continued growth in device performance, the increasing application of heterogeneous structures such as GP GPUs for acceleration, the increase in total system scale measured in terms of number of concurrent threads, and the emergence of a new generation of pGAS (Partitioned Global Address Space) programming languages.

A major effort is underway through a number of industry and academic projects to develop extensions to Unix and Linux operating systems to manage the multi-core resources of the next generation systems. The initial starting point is a kernel on every core. This requires cross operating system transactions through the I/O name space for even the simplest of parallel processing. Operating system kernels that can manage all the cores on a single socket or even multiple sockets are being developed. This is similar to earlier work on SMPs and enterprise servers of previous generations. But they must now become ubiquitous as essentially all computing systems down to the single user laptop is or will shortly become of this form. One of the key challenges is the implementation of light weight threads. Conventional Unix P-threads are relatively heavy weight providing strong protection between threads but requiring a lot of overhead work to create and manage them. User multi-threaded applications will require threads with a minimum of overhead for maximum scalability.

A second evolutionary trend is in node or core **virtualization**. As a wider range of microarchitecture structures and instruction sets are being applied with systems taking on a diversity of organizations as different mixes of processor types and accelerators are being structured. Virtualization provides a convenient and standardized interface for portability with the assumption that the virtualization layer is built to optimally employ the underlying system hardware. Where this is not feasible, separate interface libraries are being developed to make such resources at least accessible to users should they choose to take advantage of them.

The community as a whole is converging in a set of functionalities at the middleware level. These are derived from a number of usually separately developed software packages that have been integrated by more than one team in to distributions that are ever more widely used, especially across the cluster community. However, the individual pieces are also being improved for greater performance, scalability, and reliability, as well as advanced services. This is particularly true in the area of schedulers at the system level and microarchitecture level.

6.9 Programming Models and Languages

To support the extreme parallelism that this report predicts will be required to achieve Exascale computing, it is necessary to consider the use of programming models beyond those in current use today as described in section 4.3. The road map for programming models and languages is driven by two primary factors: the adoption rate in the application community of application developers and the investment available to develop and deploy the underlying compiler and run times required by new languages and models. Both pose significant challenges which must be overcome. Although the uncertainties posed by these factors makes a precise road map difficult, we describe the most likely path in this section.

6.9.1 The Evolution of Languages and Models

In the past decades, there have been numerous attempts to introduce new and improved programming languages and compilers targeted both at mainstream and high-end computing. While several of these attempts have succeeded, most have failed. To understand what leads to the success and failure of new approaches in general, it is instructive to examine their lifespan.

Programming languages and models are generally initiated by one of two paths: **innovation** and **standardization**. In the innovation path, there is generally a single organization which is proposing some new concept, produces a tool chain, attempts to attract users (usually themselves first), and aims to get a significant user base. This is important because the costs of implementing infrastructure are high and must be justified over a large (at least potential) user community.

The standardization path is quite different. It involves a group of organizations coming together either to combine related language and programming model concepts or amend existing standards. Here, the innovation introduced is far less than the other approach, but the community of users is almost guaranteed and often a commitment exists to the required tools at the outset.

Regardless of the path taken, these efforts have many common elements: they start with a period of rapid change in which experimentation is performed but in which the user community is small; then there is a crucial period where the rate of change slows and the user community is either attracted to the ideas and a critical mass is established, or the project fades away; and finally, for successful projects, there is a life-cycle support in which only very minor changes are made due to pressure from the user community.

In addition to these factors, a number of others influence success or failure. For example, the ideas represented in Java had been proposed numerous times before, but until they were popularized in C++ object-oriented programming, these new ideas did not achieve critical mass. Finally it should be noted that the path from innovation to adoption is quite long, usually about a decade.

6.9.2 Road map

Here we list and describe the path that specific language and model efforts will most likely take in the next 5-7 years without a significant uptick in funded research and development efforts.

The major incumbent models and languages in the high-end world will continue to remain the dominant mechanism for high-end computing. MPI will continue to dominate the message passing models and will certainly be widely deployed. It will most likely evolve to include some new features and optimizations. For a class of applications and users, this will be viewed as sufficient. In the various non-message passing realms, the dominant influence on language will be the arrival of significant multi-core microprocessor architectures. It is clear that the non-high-end community needs to develop models here if the architecture is to be successful. It appears somewhat likely that some current language efforts which involve either threaded approaches (Cilk, pthreads etc.) or pGAS (Chapel, Co-Array Fortran, Titanium, UPC, X10) will influence the course.

There is also a trend towards better language and model support for accelerator hardware. For example, Cuda (NVidia) and BTBB (Intel) are competing to support the graphics processor accelerators. SIMD extensions remain popular. And support for memory consistency models, atomic memory operations and transactional memory systems are being investigated to support better shared memory program synchronization. We also see a trend to hybridization of many of the above approaches. While some of these are laudable for either local node performance or synchronization, in their current form most of these mechanisms seem too specialized for widespread adoption.

Chapter 7

Strawmen: Where Evolution Is and Is Not Enough

This chapter attempts to take the best of known currently available technologies, and project ahead what such technologies would get us in the 2013-2014 time-frame, where technology decisions would be made for 2015 Exascale systems. The results of these projections will provide the basis for the challenges that await such developments.

7.1 Subsystem Projections

To set the stage in terms of general feasibility and potential challenges, a series of short projections were made of different aspects of Exascale systems, and are documented in the following sections. While in most cases the focus was on the data center class of Exascale systems, the results are for the most part directly translatable to the other two classes.

The order of presentation of these results are more or less in a “bottom-up” correspondence to parts of the generic architectures presented in Chapter 4.

Since the goal of this study was not to design Exascale systems, many of these brief studies do not focus on point designs but on generic numbers that can be scaled as appropriate to match desired application needs. Thus memory capacity is discussed in terms of “per petabyte,” and power dissipation expressed in terms of system energy expended per some unit of performance and some baseline implementation.

7.1.1 Measurement Units

Much of the baseline numerics below were chosen to be reasonable but rounded a bit to make calculations easy to follow. Thus for example, we use as a baseline of performance an **exa instruction processed to completion** (1 **EIP**), that corresponds to executing and retiring 10^{18} more or less conventional instructions. When we wish to denote the completion of 1 EIP in 1 second, we will use (in analogy to MIPS) the term “EIPs” (Exa Instructions Processed per second) Again, the term EFlops refers to the completion of 10^{18} floating point operations in a second, and is a different measure than EIPs.

In most typical programs perhaps 40% of such instructions reference memory; we will round that up to 50%. Thus for each EIP executed per second, perhaps 0.5×10^{18} distinct references to memory are made from the core itself into the memory hierarchy.

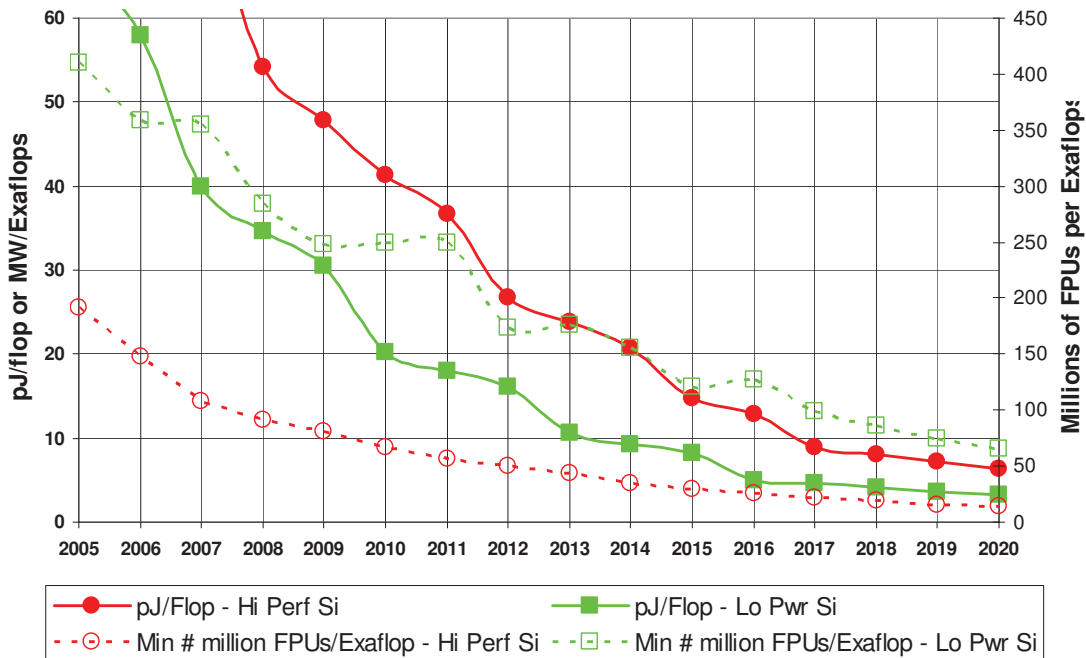


Figure 7.1: Projections to reach an Exaflop per second.

Further, since in most cached hierarchies only some percentage of such memory references actually go off to real memory, and this percentage varies both on the size of the caches and the application, we will baseline a unit of cache miss at 2%. Thus for each EIP executed per second, and each 2% of miss that the application actually experiences, there are about $0.5 \times 10^{18} \times 0.02 = 10^{16}$ distinct references that must reach main memory.

Finally, assuming a classical computer ISA where individual floating point instructions trigger one floating point operation, and that the percentage of such instructions in a program is on the order of 10% (again to make the math simple to follow), then achieving a sustained 1 EFLOP/s (10^{18} flops) requires 10 EIP/s, which in turn requires 10^{17} real memory accesses for each 2% of overall cache miss exhibited by the application. Thus a 20% miss rate for a sustained 1 EFLOP/s application with a 10% floating point mix would see on the order of 10^{18} distinct memory accesses per second (or 1 exa accesses per second).

For reference Murphy[109] analyzes a range of high end applications where floating point mixes range from 6 to 40%, and miss rates out of even very large caches can exceed 60%.

In terms of the relationship between energy and power, we note that numerically, if computing some operation takes X pJ of energy, then computing 10^{18} of them in one second consumes X MW of power.

7.1.2 FPU Power Alone

While this study does not equate exaflops to Exascale computing, understanding whether or not, and when, conventional silicon can provide an exaflop per second within the desired power limits is a useful exercise to bound the rest of the discussion. Using the ITRS data from Section 6.2.1, Figure 7.1 projects the energy expended for a single flop as a function of time for both high performance

Core	L1:I+D (KB)	FPU	Power	Tech	Area	Vdd	Clock (GHz)	Native pJ/Cycle	90nm pJ/Cycle	90nm Area
Niagara-I	24	No	2.06	90	11.9	1.2	1.2	1719	1444	11.9
Niagara-II	24	yes	3.31	65	12.4	1.1	1.4	2364	3274	23.9
MIPS64	40	No	0.45	130		1.2	0.6	750	436	0.0

Figure 7.2: Energy per cycle for several cores.

silicon and low power variants. The baseline for this estimate is a 100 pJ per flop FPU built in 90 nm technology, as discussed in [43] .

As noted before, numerically an X pJ per operation is the same as X MW for an exa operation of the same type per second. Thus, it isn't until 2014 that the power for flops alone drops below 20MW for conventional high performance CMOS. In fact, it isn't until 2020 that the power is low enough to even conceive of enough margin to account for other factors such as memory or interconnect.

The story for low power CMOS is better. In the 2013-2014 timeframe FPUs alone would dissipate in the order of 6-7 MW, enough to at least leave some space for the other functions.

A second key observation to make from this data is the number of FPUs that are needed at a minimum to reach the exaflop per second limit. The dotted two curves in Figure 7.1 give an estimate of this for both types of logic, assuming that the implementations are run at the maximum possible clock rate (where power density becomes a significant chip problem). High power silicon needs only about 50 million copies to get to an exaflop per second, but this is assuming a 20+GHz clock and a power dissipation per unit area of almost an order of magnitude greater than today. The low power silicon's maximum rate is only in the 5-6GHz range, but now requires at a minimum 200 million copies, and still has a power dissipation density exceeding today.

The clear conclusion is that if we are to rely on silicon for floating point functions in 2013-2014, then it has to be using the low power form, and even then there are problems with power density (requiring lowering the clock) and in the huge concurrency that results (which gets even either worse at lower clock rates). The strawmen discussed later in this chapter assume both.

7.1.3 Core Energy

Clearly FPU energy alone is not the only processing-related logic that will probably be present in an Exascale system. To get a handle on how large such energies might be, Figure 7.2 takes some data on existing 64 bit cores[93][159][110], and computes an energy per clock cycle, both for the real base technology and normalized to 90 nm (the 2005 base in the prior charts). The results reveal both a large variation in energy per cycle, and a significant multiple over what the FPU assumed above would be.

The Niagara rows are especially revealing. Both are multi-threaded, which implies that there is a bigger register file, and thus a larger energy component for reading registers, than for classical single-threaded cores. Whether or not this accounts for the approximately 3X difference from Niagara I to the MIPS64 line is unclear, especially considering the MIPS64 is a dual issue microarchitecture, while that for the Niagara is only single issue.

In addition, the Niagara I does not have any integrated FPU, while that for the Niagara II has not only an FPU but a whole separate graphics pipeline. When normalized, the difference in energy of about 1300 pJ is significantly more than that for our assumed FPU, even if we subtract out about 30% of the measured power to account for static leakage.

The bottom line from this discussion is that it is not just the FPUs that need to be considered,

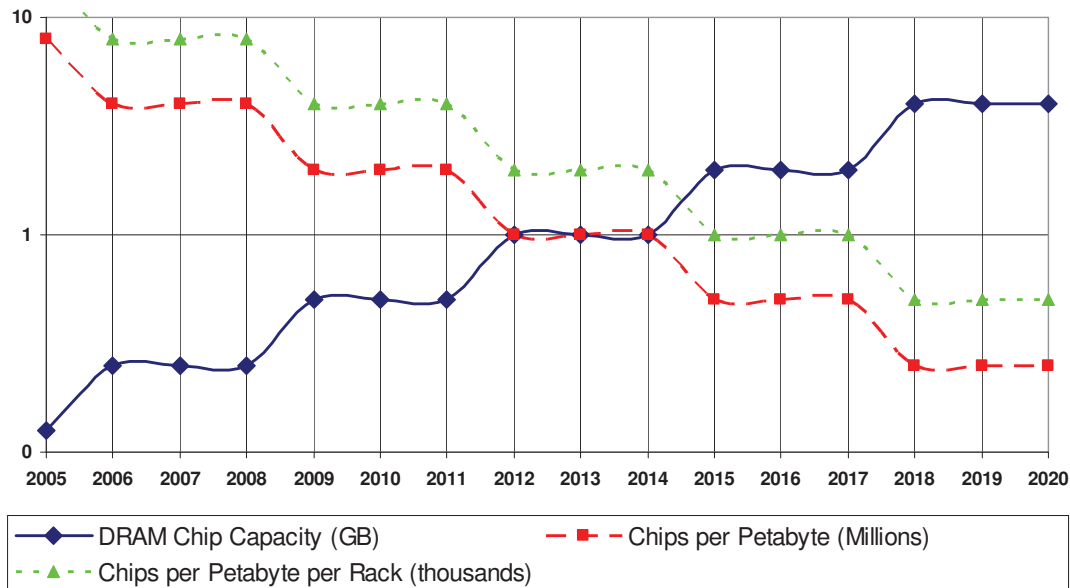


Figure 7.3: DRAM as main memory for data center class systems.

even for flop-intensive applications, and that if we are to minimize overall energy, it is essential to devise microarchitectures where the overhead energy for fetching instructions, decoding them, fetching operands, and managing their retirement is both as low as possible, and spread out over as many flops as possible.

7.1.4 Main Memory from DRAM

The needed capacity of the “main memory” level of the memory hierarchy of Chapter 4 has proven to be one of the most volatile requirements of Exascale applications, with “proof of concept” systems possible with just a few petabytes of storage, but more demanding and perhaps significant applications (that is those that would justify such a machine) needing in the hundreds of petabytes. The following sections walk through various aspects of implementing such main memory using today’s DRAM technology as it is projected to mature through time.

7.1.4.1 Number of Chips

Figure 7.3 includes a projection for the capacity of high volume commodity DRAM memory chips through time. In the 2013-2014 time frame this is about 1 GB per chip.

The second line in this figure is the number of chips needed to reach a petabyte. In 2013-2014 this is about a million chips per PB. Thus, if the actual application requirement is for 100PB of main memory, this would take 100 million commodity DRAM chips. This implies a real need for a denser packaging of DRAM than what is used today.

The third line on the graph is the number of chips per rack, assuming 500 racks in the system. Again, this is per PB, so if the actual need was for 100PB, then in 2013-2014 we would need on the order of 200,000 commodity DRAM chips to be packaged in each rack. For reference, today’s supercomputers may house at most a few thousand such chips per rack.

Assuming a FIT rate of 10 per billion hours per chip, such volumes of chips quickly translate into a mean time between memory chip failures of between 1 and 100 hours, depending on the overall memory capacity.

Note that no ECC or other redundancy is included in these numbers, so some growth in chip count for a realistic system should be expected (perhaps 12% for a standard SECDED code on each 64 bit word). While increasing the overall system FIT rate from a component perspective, such density increase would improve the MTBF of the memory part of the system.

7.1.4.2 Off-chip Bandwidth

As discussed in Section 7.1.1, a total number of distinct accesses to main memory of between 10^{16} (10 peta accesses) and 10^{18} (1 exa access) per second as an application-driven goal is not unreasonable. Assuming a conventional DRAM-like interface to these chips, each access results in the transfer of between 8 and 32 bytes of data (a word to a cache line) across some set of chip boundaries - not counting ECC. Rounding up a bit, this translates into an aggregate transfer rate off of the memory chips of between 0.1 and 32 exabytes per second.

We note that these aggregate rates are independent of the memory capacity. Thus on average the data rate per chip is this number divided by the number of chips. For 1 PB (a million chips), this translates to 100 to 32,000 GB/s - per chip! These numbers are far in excess of any projected commodity memory chip signalling protocol. For 100 PB (100 million chips), this reduces to a mere 1 to 320 GB/s per chip.

We also note that today's memory parts are typically perhaps a byte wide, and to access larger entities, multiple chips are ganged and accessed in parallel. This means that the address and control bits that must be transferred per access must be repeated per chip. If this address and control information averages 32 bits per access, then if it must be broadcast identically to 8-10 chips in parallel, there are upwards of 300+ extra bits that must be transferred per access across chip boundaries. This has the potential to almost double the overall bit transfer rate to and from a main memory made of DRAM.

Finally, we note that going to a denser memory technology makes the off-chip bandwidth requirements even worse, since there are fewer chips for any particular memory capacity.

7.1.4.3 On-chip Concurrency

For this section an **independent memory bank** is that part of a memory circuit that can respond to an address and deliver a row of data as a result. This includes row decoders, memory mats, and sense amplifiers, but need not include anything else. Thus an **active bank** is one that is responding to a separate memory request by activating a memory mat, and reading out the data.

For reference, today's memory chip architectures support at best about 8 such independent banks.

Understanding how many such banks need to be capable of independent access on a typical memory chip is thus important to determine how much "overhead logic" is needed to manage the multiple banks, and how much power is dissipated in the memory mats where the data actually resides.

An average for the number of accesses per second that must be handled per chip can be estimated by taking the total number of independent memory references per second that must be handled, and dividing by the number of memory chips. The first is a property of the application and the caching hierarchy of the processing logic; the second is related directly to the capacity of the memory system. Figure 7.4(a) lists this for a variety of memory reference rates (where 10^{16} to 10^{18} is what

		Main Memory Capacity (in PB)			
		1	10	100	1000
Accesses/sec	1.E+15	1	0.10	0.01	0.001
	1.E+16	10	1	0.10	0.01
	1.E+17	100	10	1.00	0.10
	1.E+18	1000	100	10	1
	1.E+19	10000	1000	100	10

(a) References per chip (in billions)

		Main Memory Capacity (in PB)			
		1	10	100	1000
Accesses/sec	1.E+15	10	1	0.1	0.01
	1.E+16	100	10	1	0.1
	1.E+17	1000	100	10	1
	1.E+18	10000	1000	100	10
	1.E+19	100000	10000	1000	100

(b) Active Banks per chip

Figure 7.4: Memory access rates in DRAM main memory.

was discussed previously) and the spectrum of main memory capacities discussed for applications of interest.

To convert this into a number of independent banks requires estimating the throughput of a single bank, that is, the maximum rate that a single bank can respond to memory requests per second. For this estimate, we assume a bank can handle a new request every 10 ns, for a throughput of 10^8 references per second per chip. (This is a bit of an optimistic number, given current technology, but not too far off.) Figure 7.4(b) then uses this number to compute the number of active banks for each of the prior configurations. As can be seen, if main memory access rates exceed 10^{16} per second (on the low side of what was discussed above), then it isn't until memory capacities exceed 100 PB (100 million chips) that the average number of active banks drops down to what is close to today.

7.1.5 Packaging and Cooling

The degree of difficulty posed by the packaging challenge depends on the memory bandwidth requirements of the eventual system. In this section we use as a basis the strawman machine as discussed in Section 7.3. The lower end of the requirement, 44 GBps from the CPU to each of 16 DRAM die can most likely be satisfied with conventional high-end packaging. The high-end requirement of 320 GBps to each of 32 DRAMs is well beyond the means of any currently available packaging technology. Some embedded applications were investigated to obtain another perspective on the possible size requirement. As discussed elsewhere in this document, the usual requirement of 1 Byte per second and 1 Byte of memory for every FLOPS, would require larger scales of memory system but would have significant power issues. A system anywhere near this scale would require significant advances in interconnect and packaging. In particular, advances in 3D system geometries would be required.

An advance in 3D packaging also presents an opportunity to use geometry to reduce power consumption. With a conventional packaging approach, aggressive scaling of interconnect power

would permit memory-CPU communications at an energy cost of around 2 pJ/bit. On the other hand, some 3D integration technologies, would permit power levels to approach 1-20 fJ/bit, depending on the length of run of on-chip interconnect required. Even at the low end bandwidth of 16 x 44 GBps, this represents a power savings of around 10 W per module, which could be more productively used for FPU's or additional memory capacity.

As well as provisioning interconnect, packaging also plays roles in power and ground current distribution, noise control (through embedded capacitors), heat removal and mechanical support, so as to ensure high reliability. The simultaneous roles of current delivery and heat removal create a geometric conundrum as the high power areas of the chip need lots of both at the same time. This requirement leads to the usual solution in single-chip packaging of using the front-side of the chip for current delivery and the backside for cooling. Such arrangements are not easily scaled to 3D integration as one side of the chip has to be used for the 3D mating. As a result, the typical state of the art expected for circa 2015 would be a 3D chip stack mating a small number (3-4) of memory die to the face side of a CPU. While the back-side of the CPU chip is used for cooling, extra through-silicon vias are built into the memory stack in order to deliver current to the CPU. Stacks beyond 4-5 die are not expected, due to the combined problem of sufficiently cooling the interior die (through the hot CPU), and providing current delivery through the memory die stack.

Some options that could be pursued are summarized in Figures 7.5 to 7.7. The first suggestion (distributed 3D stacks) simply avoids this problem by portioning the CPU amongst a number of smaller memory stacks, and integrating these 3DICs using a silicon carrier. However, the cost of this is a dramatic reduction in inter-core bandwidth, once they cross chip boundaries. This is unlikely to be acceptable in many applications.

The second suggested option is referred to as an "advanced 3D IC," that is a 3D IC with package layers incorporated into it, in the 3D stack. These package layers are used for current distribution and heat removal from die within the stack. The result would be a heterogeneous chip stack, tens of units high. Considerable challenges exist in this option. The technology to build such a stack with high yields does not exist today.

A variant of the "advanced 3D IC" solution is a more package-centric solution that allows 3D integration of multiple chip stacks. This could be done using a combination of chip-on-wafer and wafer-on-wafer technologies. However, it would still require considerable technology investment.

A variant of Sun's proximity connection could also be adapted to solve this problem. However, a combination of technologies could be used for interconnect, rather than just relying on capacitive coupling. As discussed in Section 6.6, Sun's original scheme implicitly requires current distribution and heat removal through **both** sides of the 3D package, which of course is difficult. However, supplementing their approach with through-silicon vias could relieve this situation and provide an interesting solution.

Another approach would be to use something like Notre Dame's "quilt packaging" [15] to provide high density edge connections, on a tight sub-20 μ m pitch. In their solution, all the chips are face-up, so the two-sided power delivery and removal problem is easily solved. However, one challenge with both of these solutions revolves around the fact that they are both edge I/O approaches. Numerous, long power-hungry on-chip wires would be required to join peripheral memories to interior cores.

Several groups have investigated edge-mounting of die on the planar surface of another die. This leads to a memory edge mounting solution, such as shown in Figure 7.7. There are several complications with this solution. It does not directly solve the two-side power delivery/removal problem. Also, the memories would require long, power-hungry on-chip traces.

Finally, it should be realized that advances in areas outside of packaging could simplify the creation of an Exascale 3D solution. In particular, if efficient voltage conversion could be incorporated within the chip stack, then the "two-sided problem" is greatly simplified. Delivering 100 A at 1 V

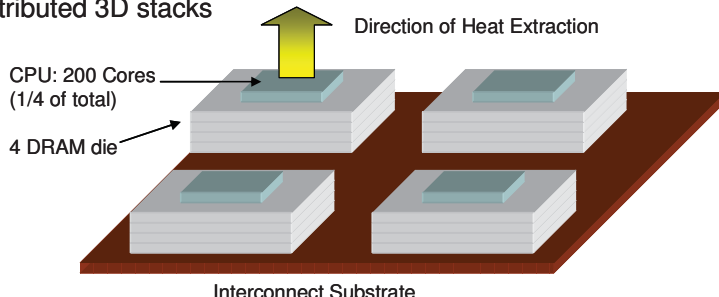
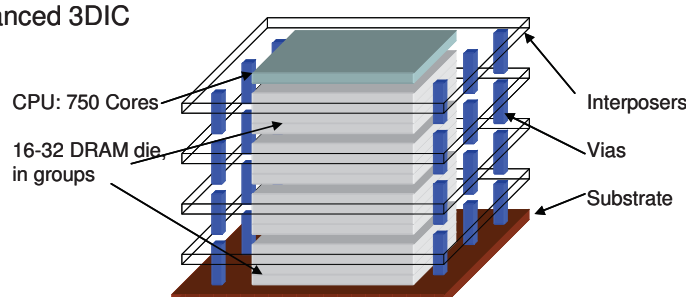
Approach	Comments
<p>Distributed 3D stacks</p>  <p>CPU: 200 Cores (1/4 of total)</p> <p>4 DRAM die</p> <p>Direction of Heat Extraction</p> <p>Interconnect Substrate</p>	<p>Distribute CPU across multiple memory stacks</p> <p>Assumes sufficient inter-stack bandwidth can be provided in substrate</p> <p>Likely to detract from performance, depending on degree of memory scatter</p>
<p>Advanced 3DIC</p>  <p>CPU: 750 Cores</p> <p>16-32 DRAM die in groups</p> <p>Interposers</p> <p>Vias</p> <p>Substrate</p>	<p>Incorporate interposers into a single 17-33 chip stack to help in power/ground distribution and heat removal.</p> <p>Assumes Through Silicon Vias for signal I/O throughout chip stack</p>

Figure 7.5: Potential directions for 3D packaging (A).

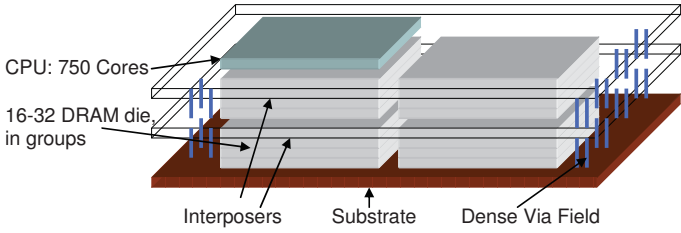
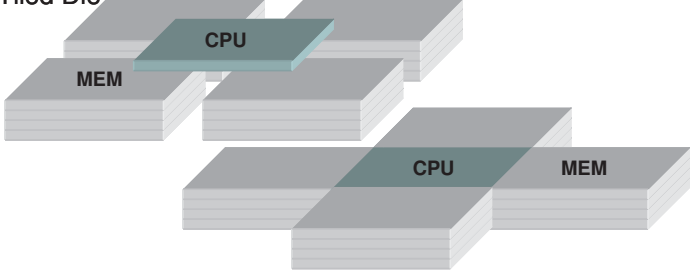
Approach	Comments
<p>Advanced 3D Package</p>  <p>CPU: 750 Cores 16-32 DRAM die, in groups Interposers Substrate Dense Via Field</p>	<p>To avoid complexity of a 33-chip stack, this approach, users the interposers for high density signal redistribution, as well as assisting in power/ground distribution and heat removal.</p> <p>Requires a planar routing density greater than currently provided in thin film carriers.</p>
<p>Tiled Die</p>  <p>CPU MEM CPU MEM</p>	<p>Use proximity connection or Through Silicon Vias to create memory bandwidth through overlapping surfaces.</p> <p>OR</p> <p>Tile with high bandwidth edge interfaces, using quilt packaging or a an added top metal process. (Note, impact on latency and I/O power).</p>

Figure 7.6: Potential directions for 3D packaging (B).

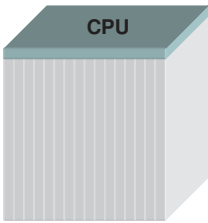
Approach	Comments
<p>Edge Mounting</p>  <p>CPU</p>	<p>Stack memory die on top of CPU using a 3D quilt process or functional equivalent.</p> <p>Requires considerable technological innovation</p>
<p>Orthogonal Solutions</p> <p>On-die or in-stack voltage conversion Embedded liquid cooling mm-thick high capacity heat pipes</p>	<p>These solutions provide ways to alleviate some of the difficulties in the solutions above.</p>

Figure 7.7: Potential directions for 3D packaging (C).

	PB of Main Memory				
	0.006	0.5	3.6	50	300
Scratch Storage					
Capacity (EB)	1.2E-04	0.01	0.15	2	18
Drive Count	1.0E+01	8.3E+02	1.3E+04	1.7E+05	1.5E+06
Power (KW)	9.4E-02	7.8E+00	1.2E+02	1.6E+03	1.4E+04
Checkpoint Time (sec)	1.2E+03	1.2E+03	5.8E+02	6.0E+02	4.0E+02
Checkpoint BW (TB/s)	5.0E-03	4.2E-01	6.3E+00	8.3E+01	7.5E+02
Archival Storage					
Capacity (EB)	0.0012	0.1	7.2	100	600
Drive Count	1.0E+02	8.3E+03	6.0E+05	8.3E+06	5.0E+07
Power (KW)	9.4E-01	7.8E+01	5.6E+03	7.8E+04	4.7E+05

Table 7.1: Non-memory storage projections for Exascale systems.

is a lot harder than delivering 1 A at 100 V. Similarly, releasing one side from a cooling function provides a similar improvement. For example, incorporating silicon micro-channel cooling into the chip stack removes the need for using one side for heat removal.

On the topic of cooling, it was presumed for this study that at the system level, the computing stack would be air-cooled. Large scale deployment of water chillers and circulating coolants does not lend itself to embedded solutions in war fighting craft and vehicles. However, this global issues does not prevent the local solution from using liquid phase solutions locally. As long as the liquid does not require replenishment, then such local solutions might be reasonable. There are severe challenges in such solutions though. Leaks are still a problem in liquid cooling, and a leak-free technology would be a worthwhile investment. Also, techniques to circulate and pump the coolant are needed on the physical scale of a small 3D package. Similarly, local heat exchangers would be needed if local liquid cooling solutions are to be used. Heat pipes provide an excellent way to use liquid phase cooling locally without mechanical complexity. Advances in the capacity of thin heat-pipe like solutions would be welcome in an Exascale computer.

7.1.6 Non-Main Memory Storage

Using the numbers from Section 5.6.3, especially as articulated in Table 5.1, and the projections from Section 6.4.1, Table 7.1 summarizes numbers for scratch and archival storage systems using disk technology from 2014. Consumer-grade disks are assumed because of the need for sheer density. As before, these numbers do not include either additional space for RAID or for controllers and interconnect, and as such represent a lower bound.

The Checkpointing time and bandwidth assume that *all* of the drives in the Scratch system are accepting data concurrently, and at their maximum projected data rates. This is not reasonable, especially for the larger systems.

The numbers given here are given as a function of main memory. As such, the options were chosen to match different sizes of systems:

- The 6TB column corresponds to the main memory capacity of a single rack of the aggressive strawman of Section 7.3.
- The 0.5PB column corresponds to the “sweet spot” for a departmental system as suggested in Table 5.1.

- The 3.6PB column corresponds to the main memory capacity of the complete exaflops aggressive strawman of Section 7.3.
- The 50PB column corresponds to the “sweet spot” for a data center class system as suggested in Table 5.1.
- The 300PB column corresponds to a data center class system that has the same memory to flops ratio as today’s supercomputers.

As can be seen, the scratch disk counts for both classes of systems are not unreasonable until the main memory capacities approach ratios typical of today, such as 1.4M drives and 14 MW for a 0.3 to 1 byte to flop ratio at the data center scale.

Checkpointing time across all systems is in the realm of 10 to 20 minutes (under the optimistic assumptions of all drives storing at max rate). When stretched to more realistic times, this implies that checkpointing times may be in the region of the MTBF of the system. As discussed in Section 6.7.4, this may render the data center class systems effectively useless, indicating that there may be a real need in finding a memory technology with higher bandwidth potential (especially write rates) than disk. Variations on Flash memory may offer such a capability at competitive densities and power.

The archival numbers of course are higher, and probably reach the limits of rational design at 3.6PB main memory for the data center class strawman of Section 7.3, where 600,000 drive at close to 6 MW are needed.

7.1.7 Summary Observations

The above short projections lead inescapably to the following conclusions:

1. If floating point is key to a system’s performance, and if CMOS silicon is to be used to construct the FPU’s, then, for the data center class, to have any hope of fitting within a 20 MW window, the low operating power variant must be chosen over today’s high performance design. This is also true of the other two classes, as the use of high performance logic results in power densities that exceed their form factor limits.
2. Energy for the processor core must be very carefully watched, lest power exceed that for the basic operations by a large factor. This implies that microarchitectures must be designed and selected for low energy per issued operation.
3. Unless at best a very few PB of main memory is all that is needed, then DRAM technology by itself is inadequate in both power and capacity.
4. To reduce the bit rates and associated energy of transfer to manage the address and control functions of an access, each DRAM chip needs to be rearchitected with a wide data path on and off chip, low energy per bit interfaces, and many potentially active banks.
5. Unless memory capacity gets very large, then using DRAM for main memory requires that the architecture of the DRAM provide for far more concurrently active mat access (regardless of word widths) than present today.



Figure 7.8: A typical heavy node reference board.

7.2 Evolutionary Data Center Class Strawmen

To baseline where “business-as-usual” might take us, this section develops two strawmen projections of what evolution of today’s leading edge HPC data center class systems might lead to. This will stand in contrast to both the trend lines from Section 4.5 (that assumes technology will advance uniformly at the rate that it has for the last 20 years, but with the same architectural and packaging approach) and from the aggressive strawman of Section 7.3 (that assumes we can choose the best of current technologies in a relatively “clean-sheet” approach).

The architectures used as the two departure points for this study are “heavy node” Red Storm-class machines that use commodity leading edge microprocessors at their maximum clock (and power) limits, and “light node” Blue Gene/L class machines where special processor chips are run at less than max power considerations so that very high physical packaging densities can be achieved.

7.2.1 Heavy Node Strawmen

Machines such as the Red Storm and its commercial follow-ons in the Cray XT series assume relatively large boards such as pictured in Figure 7.8[90]. A few dozen of these boards go into an air-cooled rack, and some number of racks make up a system.

7.2.1.1 A Baseline

A “baseline” board today contains

- multiple (4) leading edge microprocessors such as from Intel or AMD, each of which is the heart of a “node.” A substantial heat sink is needed for each microprocessor.
- for each node a set of relatively commodity daughter memory cards holding commodity DRAM chips (FB-DIMMs as a baseline).
- multiple specialized router chips that provide interconnection between the on-board nodes and other boards in the same and other racks.

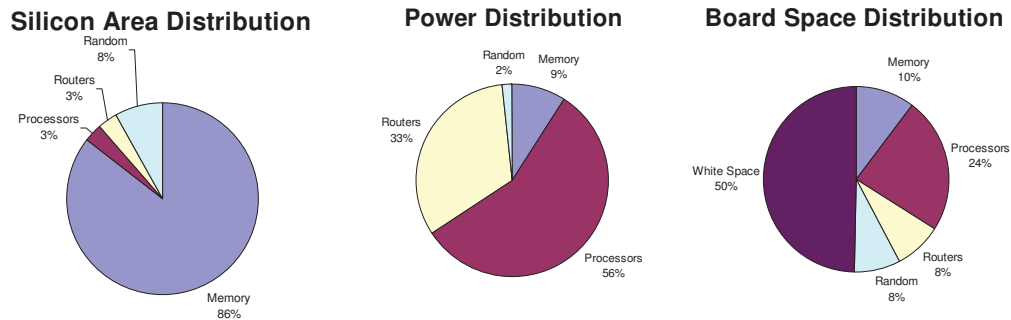


Figure 7.9: Characteristics of a typical board today.

- an assortment of support circuitry.

Very often the chip set associated with each node is referred to as a **socket**, since the single biggest physical entity on such a board is the microprocessor and its heatsink as pinned in its socket.

Figure 7.9 then summarizes the approximate characteristics of such a board as to where the silicon is used, which chips dissipate what part of the power, and how the board space is occupied.

In terms of a system baseline, we assume 4 sockets per board and nominally 24 boards per rack. A 2.5 MW system consisting of 155 racks would contain 12960 sockets for compute and 640 sockets for system, or one system socket for approximately every 20 compute sockets.

At 40 TB of storage and an R_{peak} of 127 Tflops, the baseline has a ratio of about 0.31 bytes per peak flop.

7.2.1.2 Scaling Assumptions

Extrapolating the same architecture forward into the future, what kind of parts are on such a physical board would not change much: a relatively constant amount of real estate is needed for each of the major subassemblies, especially the heat-sinked parts. To a first order what might change is the number of nodes/sockets on a board (fill the white space), the complexity of the microprocessor chips (succeeding generations would include more cores), the density of memory chips on the memory daughter cards will increase, and perhaps the number of memory cards (if performance per node increases faster than memory density increases). Thus our assumptions about future versions of such a system are as follows:

- The die size for the microprocessor chip will remain approximately constant, meaning that the number of cores on each microprocessor chip may grow roughly as the transistor density grows, as pictured in Figure 4.3. We do not account for relatively larger L3 caches to reduce pressure on off-chip contacts.
- V_{dd} for these microprocessors will flatten (Figure 4.7 and 6.2), as will maximum possible power dissipation (Figure 4.10), which means that the maximum clock rate for these chips will approximately flatten as discussed in Section 6.2.1.5.
- On a per core basis, the microarchitecture will improve from a peak of 2 flops per cycle per core in 2004 to a peak of 4 flops per cycle in 2008, and perhaps 8 flops per cycle in 2015.

- The system will want to maintain the same ratio of bytes of main memory to peak flops as today, and to do this we will use whatever natural increase in density comes about from commodity DRAM, but coupled with additional memory cards or stacks of memory chips as necessary if that intrinsic growth is insufficient.
- The maximum number of sockets (i.e. nodes) per board may double a few times. This is assumed possible because of a possible move to liquid cooling, for example, where more power can be dissipated and allowing the white space to be used and/or the size of the heat sinks to be reduced. For this projection we will assume this may happen at roughly five year intervals.
- The maximum number of boards per rack may increase by perhaps 33% again because of assumed improvements in physical packaging and cooling, and reduction in volume for support systems. For this projection we will assume this may happen once in 2010.
- The maximum power per rack may increase by at best a power of 16, to somewhere around 200-300KW. We assume this is a doubling every 3 years.
- We ignore for now any secondary storage (or growth in that storage) for either scratch, file, or archival purposes, although that must also undergo significant increases.

7.2.1.3 Power Models

We assume two power models for these extrapolations. The **Simplistic Power Scaled Model** assumes that the power per microprocessor chip grows as the ITRS roadmap has predicted, and that the power for the memory associated with each socket grows only linearly with the number of memory chips needed (i.e. the power per memory chip is “constant”). We also assume that the power associated with both the routers and the common logic remains constant.

This is clearly optimistic, since increasing the flops rate of the microprocessor most probably requires a higher number of references to memory and a higher traffic through the routers. In a real sense, we are assuming here that both memory access energy and the energy cost of moving a bit, either across a chip boundary or between racks, will improve at least as fast as the increase in flops.

In contrast, for the **Fully Scaled Power Model** we assume the microprocessor power grows as before (as the ITRS roadmap), but that both the memory and router power scale linearly with the peak flops potential of the multi-core microprocessors. This naively assumes that the total energy expended in these two subsystems is used in accessing and moving data, and that the energy to handle one bit (at whatever the rate) is constant through time (i.e. no improvement in I/O energy protocol). This is clearly an over-estimate.

Neither model takes into account the effect of only a finite number of signal I/Os available from a microprocessor die, and the power effects of trying to run the available pins at a rate consistent with the I/O demands of the multiple cores on the die.

7.2.1.4 Projections

There are a variety of ways that a future system projection could be done. First is to assume that power is totally unconstrained, but that it will take time to be able to afford a 600 rack system. The second is that we assume that we cannot exceed 20 MW. (In either case, however, we do assume a peak power per rack as discussed earlier).

	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020
Chip Level Predictions															
Relative Max Power per Microprocessor	1.00	1.05	1.10	1.10	1.10	1.10	1.10	1.10	1.10	1.10	1.10	1.10	1.10	1.10	1.10
Cores per Microprocessor	2.00	2.52	4.00	5.04	6.36	8.01	10.09	12.71	16.02	20.18	25.43	32.04	40.37	50.85	64.07
Flops per cycle per Core	2.00	2.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	8.00	8.00	8.00	8.00	8.00	8.00
Flops per cycle per Microprocessor	4.00	5.05	16.00	20.17	25.44	32.04	40.37	50.85	64.07	161.43	203.40	256.29	322.92	406.81	512.57
Power Constrained Clock Rate	1.00	0.94	1.10	0.99	0.90	0.81	0.88	0.79	0.71	0.80	0.72	0.83	0.73	0.65	0.59
Relative Rpeak per Microprocessor	1.00	1.19	4.39	4.98	5.76	6.48	8.88	9.99	11.42	32.38	36.79	52.86	58.73	66.08	75.51
Actual Rpeak per Microprocessor	9.60	11.44	42.15	47.82	55.26	62.16	85.27	95.93	109.64	310.82	353.20	507.46	563.85	634.33	724.94
ITRS Commodity Memory Capacity Growth	1.00	1.00	1.00	2.00	2.00	2.00	4.00	4.00	4.00	8.00	8.00	8.00	16.00	16.00	16.00
Required Memory Chip Count Growth	1.00	1.19	4.39	2.49	2.88	3.24	2.22	2.50	2.86	4.05	4.60	6.61	3.67	4.13	4.72
Relative Growth in BW per Memory Chip	1.00	1.00	1.00	2.00	2.00	2.00	4.00	4.00	4.00	8.00	8.00	8.00	16.00	16.00	16.00
BW Scaled Relative Memory System Power	1.00	1.19	4.39	4.98	5.76	6.48	8.88	9.99	11.42	32.38	36.79	52.86	58.73	66.08	75.51
Socket Level Predictions ("Socket" = Processor + Memory + Router)															
BW Scaled Relative per Socket Router Power	1.00	1.19	4.39	4.98	5.76	6.48	8.88	9.99	11.42	32.38	36.79	52.86	58.73	66.08	75.51
Simplistically Scaled per Socket Power	1.00	1.05	1.34	1.18	1.21	1.24	1.16	1.18	1.21	1.31	1.35	1.52	1.28	1.31	1.36
Fully Scaled Relative per Socket Power	1.00	1.12	3.53	3.15	3.71	4.28	4.88	5.63	6.67	20.77	25.15	44.44	35.32	42.11	51.64
Simplistically Scaled Relative Rpeak/Watt	1.00	1.14	3.29	4.21	4.74	5.21	7.66	8.45	9.43	24.75	27.20	34.88	45.98	50.26	55.42
Fully Scaled Relative Rpeak/Watt	1.00	1.06	1.24	1.58	1.55	1.51	1.82	1.77	1.71	1.56	1.46	1.19	1.66	1.57	1.46
Simplistically Scaled Rpeak/Watt	0.04	0.05	0.13	0.17	0.19	0.21	0.31	0.34	0.38	1.00	1.10	1.41	1.86	2.04	2.25
Fully Scaled Rpeak/Watt	0.04	0.04	0.05	0.06	0.06	0.06	0.07	0.07	0.07	0.06	0.06	0.05	0.07	0.06	0.06
Board and Rack Level Concurrency Predictions															
Maximum Sockets per Board	4	4	4	4	8	8	8	8	8	16	16	16	16	16	16
Maximum Boards per Rack	24	24	24	24	32	32	32	32	32	32	32	32	32	32	32
Maximum Sockets per Rack	96	96	96	96	256	256	256	256	256	512	512	512	512	512	512
Maximum Cores per Board	8	10	16	20	51	64	81	102	128	323	407	513	646	814	1025
Maximum Cores per Rack	192	242	384	484	1628	2050	2584	3254	4101	10331	13018	16402	20667	26036	32804
Maximum Flops per cycle per Board	16	20	64	81	204	256	323	407	513	2583	3254	4101	5167	6509	8201
Maximum Flops per cycle per Rack	384	484	1536	1936	6513	8201	10336	13018	16402	82650	104142	131218	165336	208285	262436
Board and Rack Level Power Predictions															
Max Relative Power per Rack	1	1	1	2	2	2	4	4	4	8	8	8	16	16	16
Simplistic Power-Limited Sockets/Rack	96	92	72	96	158	155	256	256	256	512	512	507	512	512	512
Fully Scaled Power-Limited Sockets/Rack	96	86	27	61	52	45	79	68	58	37	31	17	43	36	30
Simplistically Scaled Relative Rpeak per Rack	96	109	316	478	911	1001	2274	2558	2924	16577	18838	26788	30072	33831	38664
Fully Scaled Relative Rpeak per Rack	96	102	119	304	298	291	699	681	658	1197	1123	914	2554	2410	2246
System Predictions: Power Unconstrained, Gradual Increase in Affordable Racks to Max of 600															
Max Affordable Racks per System	155	200	250	300	350	400	450	500	550	600	600	600	600	600	600
Max Cores per System	29760	48441	9.6E+04	1.5E+05	5.7E+05	8.2E+05	1.2E+06	1.6E+06	2.3E+06	6.2E+06	7.8E+06	9.8E+06	1.2E+07	1.6E+07	2.0E+07
Max Flops per cycle per System	59520	96882	3.8E+05	5.8E+05	2.3E+06	3.3E+06	4.7E+06	6.5E+06	9.0E+06	5.0E+07	6.2E+07	7.9E+07	9.9E+07	1.2E+08	1.6E+08
Simplistically Scaled System Rpeak (GF)	1.0E+05	1.5E+05	5.4E+05	9.8E+05	2.2E+06	2.7E+06	7.0E+06	8.7E+06	1.1E+07	6.8E+07	7.7E+07	1.1E+08	1.2E+08	1.4E+08	1.6E+08
Fully Scaled System Rpeak (GF)	1.0E+05	1.4E+05	2.0E+05	6.2E+05	7.1E+05	7.9E+05	2.1E+06	2.3E+06	2.5E+06	4.9E+06	4.6E+06	3.7E+06	1.0E+07	9.9E+06	9.2E+06
System Power (MW)	2.5	3.2	4.0	9.7	11.3	12.9	29.0	32.3	35.5	77.4	77.4	77.4	154.8	154.8	154.8
System Predictions: Power Constrained to 20 MW															
Maximum Racks	155	200	250	300	350	400	310	310	310	155	155	155	78	78	78
Simplistically Scaled System Rpeak (GF)	1.E+05	1.E+05	5.E+05	1.E+06	2.E+06	3.E+06	5.E+06	5.E+06	6.E+06	2.E+07	2.E+07	3.E+07	2.E+07	2.E+07	2.E+07
Fully Scaled System Rpeak (GF)	1.E+05	1.E+05	2.E+05	6.E+05	7.E+05	8.E+05	1.E+06	1.E+06	1.E+06	1.E+06	1.E+06	1.E+06	1.E+06	1.E+06	1.E+06

Figure 7.10: Heavy node strawman projections.

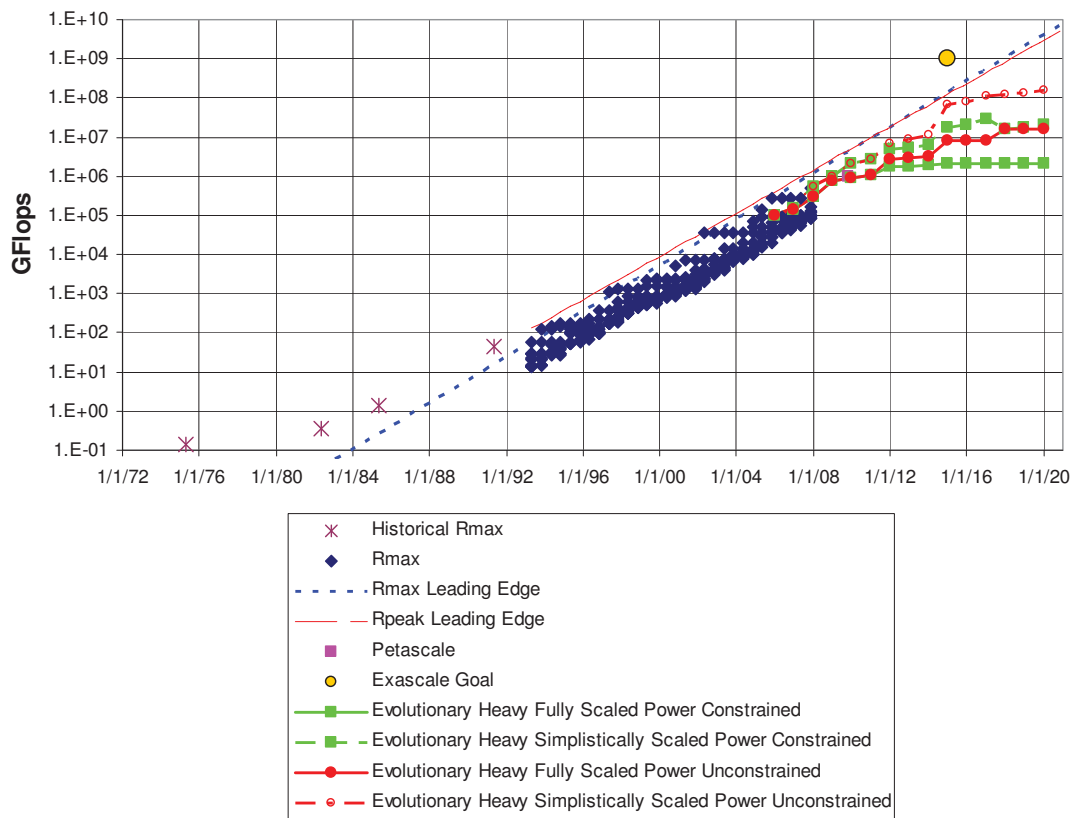


Figure 7.11: Heavy node performance projections.

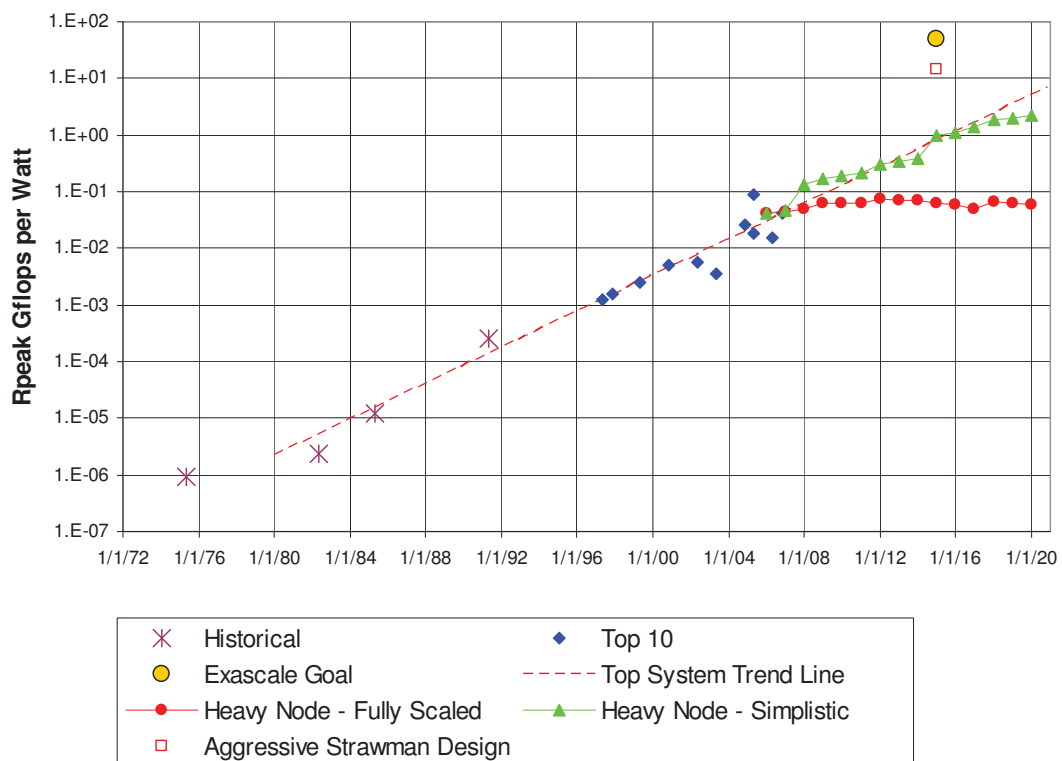


Figure 7.12: Heavy node GFlops per Watt.

Figure 7.10 summarizes some projections based on the above assumptions, all based on numbers relative to 2006, with a graphical representation of peak Linpack in Figure 7.11. In this figure, the lines marked “Power Constrained” represent a capping of power at the 20MW limit.

Figure 7.12 then also converts these numbers into gigaflops per watt, which is directly relevant to all three classes of Exascale systems.

Some specific observations include:

- Because this model used year-to-year numbers on each component, and real systems are not updated every year, there is some year-to-year “noise.”
- The clock limitations due to the power limits are significant, and greatly hamper the potential performance growth per microprocessor chip.
- The “gigaflops per watt” graph indicates that whereas the “simplistic” model seems to follow a trend line, the “fully scaled” model becomes flat, largely because the power becomes dominated not by the microprocessors but by the transport of data - to memory and to the routers. Clearly this implies a need for better off-chip protocols in terms of energy per bit, regardless of the system architecture.
- None of the overall estimates come any closer than within an order of magnitude of an exaflops, with the power constrained models running between two and three orders of magnitude too low.
- In general there is a considerable spread between the “simplistic” predictions and the “fully scaled.” Significant and more detailed projections would be needed to more accurately decide where in this range the real numbers might lie. However, both are so far off of an “exa” goal that it is clear that in any case there is a real problem.

7.2.2 Light Node Strawmen

The prior section addressed the possible evolution of systems based on leading edge high performance microprocessors where single thread performance is important. In contrast, this section projects what might be possible when “lighter weight” customized processors are used in an architecture that was designed from the ground up for dense packaging, massive replication, and explicit parallelism.

7.2.2.1 A Baseline

The basis for this extrapolation is the Blue Gene series of supercomputers, both the “/L” [48] and the “P” [147] series, with general characteristics summarized in Table 7.2. Here the basic unit of packaging is not a large board with multiple, high powered, chips needing large heat sinks, but small “DIMM-like” **compute cards** that include both memory and small, low-power, multi-core **compute chips**. The key to the high density possible by stacking a lot of such cards on a board is keeping the processing core power dissipation down to the point where only small heat sinks and simple cooling are needed.

Keeping the heat down is achieved by keeping the architecture of the individual cores simple and keeping the clock rate down. The latter has the side-effect of reducing the cycle level latency penalty of cache misses, meaning that simpler (and thus lower power) memory systems can be used. For the original Blue Gene/L core, the former is achieved by:

	Blue Gene/L[48]	Blue Gene/P[147]
Technology Node	130 nm	90 nm
FPU/Core	2 fused mpy-add	2 fused mpy-add
Cores per Compute Chip	2	4
Clock Rate	700MHz	850MHz
Flops per Compute Chip	5.6 Gflops	13.6 Gflops
Shared L3 per Compute Chip	4 MB embedded	8 MB embedded
Compute Chips per Node	1	1
DRAM capacity per Node	0.5-1 GB	2 GB
DRAM Chips per Node	9-18	20-40
Nodes per Card	2	1
Cards per Board	16	32
Boards per Rack	32	32
Nodes per Rack	1024	1024
Cores per Rack	2048	4096
R_{peak} per Rack	5.73 Tflops	13.9 Tflops
R_{max} per Rack	4.71 Tflops	
Memory per rack	1 TB	2 TB
Memory per Flop	0.17B/Flop	0.14B/Flop
Power per Compute Chip	12.9 W	
Power per rack	27.6KW	40KW
Gflops per KW	212.4	348.16
Power per Pflops	4.7 MW	2.9 MW

Table 7.2: Light node baseline based on Blue Gene.

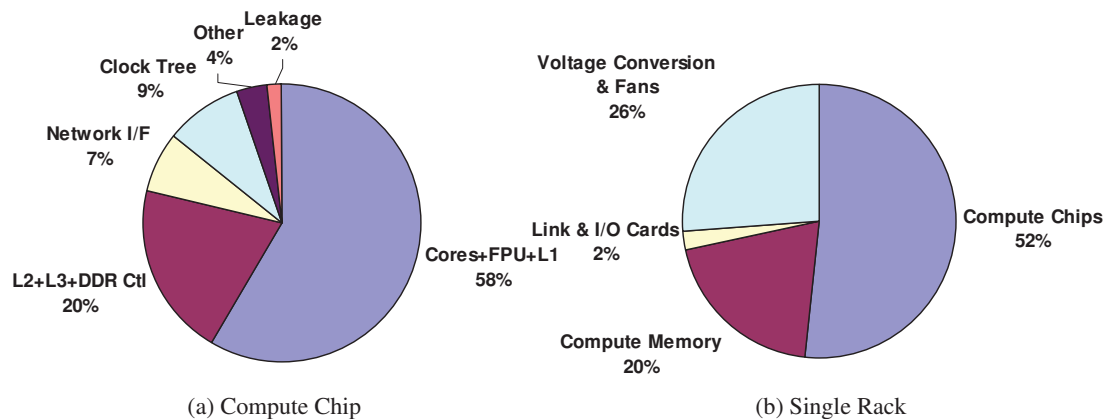


Figure 7.13: Power distribution in the light node strawman.

- a simple 7 stage pipeline,
- simple two instruction issue with 3 pipelines (load/store, integer, and other),
- a floating point unit with 2 fused multiply-adders and an extended ISA that supports simple SIMD operations at up to the equivalent of 4 flops per cycle,
- an integrated memory controller.

Several of these cores are implemented on each compute chip that also includes complete memory controllers and interface and routing functions. Thus, a complete single **compute node** consists of a compute chip and some small number of commodity DRAM chips.

One or more of these compute nodes are placed on a small circuit card akin in size to a DIMM card. Multiple such cards are then stacked on a larger board, which in turn plugs into a **midplane** holding many such boards, and several such midplanes packaged in a rack.

The current system interconnect topology is a 3D torus, with an 8x8x8 sub cube on each midplane and 4 link cards that glue the surfaces of this cube to other cubes on other midplanes. From an application perspective, this network supports a message-passing protocol, rather than a pGAS-like protocol as assumed for the heavy weight strawman of the previous section. A second interface supports a **collective network**, which is a tree-like topology that permits synchronization, barriers, and broadcasts.

In addition, on a per rack basis there are 32 compute cards and 2 I/O cards used for system access and I/O.

In terms of power, Figure 7.13 gives an approximate distribution of power by subsystem for both the compute chip and a single rack, using numbers from Blue Gene/L[20]. These numbers assume only 9 DRAM chips (512 MB) per compute node.

7.2.2.2 Scaling Assumptions

For consistency, the scaling assumptions used in Section 7.2.1.2 for the heavyweight strawman is modified only slightly:

- The die size for the compute chip will remain approximately constant, meaning that the number of cores on each chip may grow roughly as the transistor density grows, as pictured

in Figure 4.3. We do not account L3 cache sizes that grow more than linearly with core count to reduce pressure on off-chip contacts.

- V_{dd} for these microprocessors will flatten (Figure 4.7 and 6.2).
- The power dissipation per chip will be allowed to increase by 25% with every 4X gain in transistor density (this mirrors roughly the increase from Blue Gene/L to Blue Gene/P). Even at the end of the ITRS roadmap, this will still remain far under the per chip limits from the ITRS.
- On a per core basis, the microarchitecture will improve from a peak of 4 flops per cycle per core today to a peak of 8 flops per cycle in 2010.
- The system will want to maintain the same ratio of bytes of main memory to peak flops as the most recent Blue Gene/P (2 GB per 13.9 Gflops, or 0.14 to 1), and to do this we will use whatever natural increase in density comes about from commodity DRAM, but coupled with additional memory chips as necessary if that intrinsic growth is insufficient.
- The maximum number of nodes per board may double a few times. This is assumed possible because of a possible move to liquid cooling, for example, where more power can be dissipated, allowing the white space to be used, memory chips to be stacked (freeing up white space), and/or the size of the heat sinks to be reduced. For this projection we will assume this may happen at roughly five year intervals.
- The maximum number of boards per rack may increase by perhaps 33% again because of assumed improvements in physical packaging and cooling, and reduction in volume for support systems. For this projection we will assume this may happen once in 2010.
- The maximum power will be the same as projected for the heavy-weight strawman.
- The overhead for the rack for power and cooling will be the same percentage as in the Blue Gene/L.
- We ignore for now any secondary storage (or growth in that storage) for either scratch, file, or archival purposes, although that must also undergo significant increases.
- The rack overhead for I/O and power is the same percentage of compute power as the baseline, namely 33%.

For this sizing, we will ignore what all this means in terms of system topology.

7.2.2.3 Power Models

We assume the same two power models as for the heavy weight strawman (Section 7.2.1.3). The **Simplistic Power Scaled Model** assumes that the number of cores grows in proportion to the technology density increase, the power per core changes as the ITRS roadmap has predicted, and that the power for the memory associated with each node grows only linearly with the number of memory chips needed (i.e. the power per memory chip is “constant”). We also assume that the clock rate of the cores may change as governed by the maximum power dissipation allowed per compute chip. Finally, we assume that the power associated with compute chip memory and network interfaces remains constant, i.e. the energy per bit transferred or accessed improves at the same rate as the bandwidth needs of the on-chip cores increase.

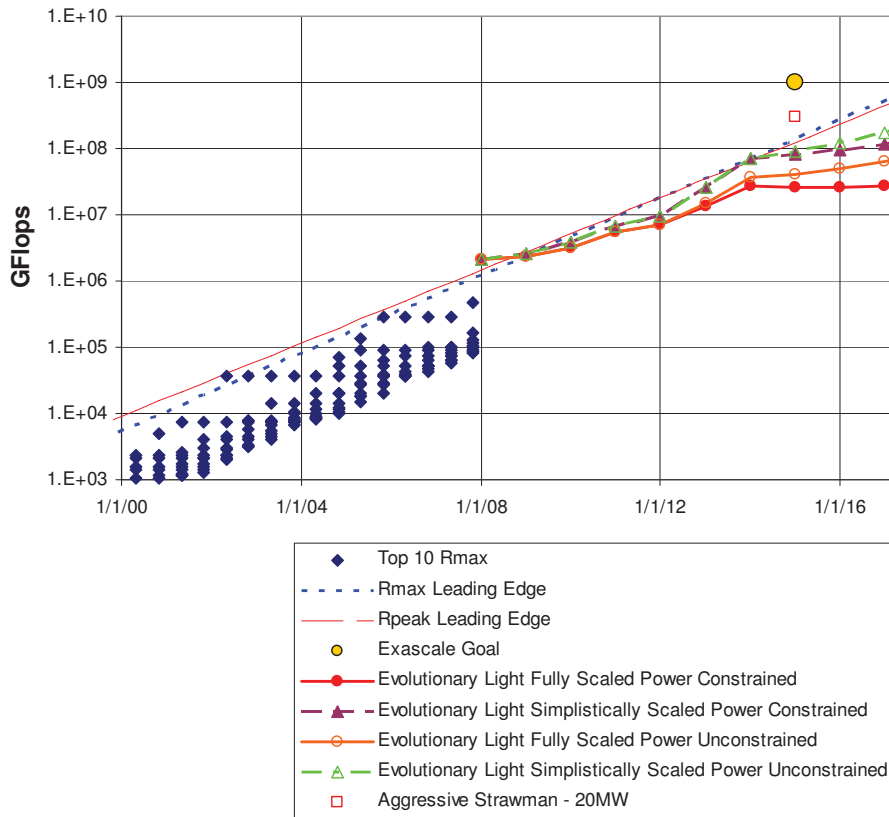


Figure 7.14: Light node strawman performance projections.

In contrast, for the **Fully Scaled Power Model** we assume the core power grows as above, but that both the memory and network interface power scales linearly with the peak flops potential of the multi-core microprocessors. This naively assumes that the total energy expended in these two subsystems is used in moving data, and that the energy to move one bit (at whatever the rate) is constant through time (i.e. no improvement in I/O energy protocol). This is clearly an over-estimate.

As with the heavyweight strawman, neither model takes into account the effect of only a finite number of signal I/Os available from a microprocessor die, and the power effects of trying to run the available pins at a rate consistent with the I/O demands of the multiple cores on the die.

7.2.2.4 Projections

Figure 7.14 presents the results from the extrapolation in the same format as Figure 7.11. One indication of the potential validity of these projections is that even though they were based on the Blue Gene/L chip, for which a great deal of data is available, the first projections seem to line up well with what is known about the Blue Gene/P chip and system.

As can be seen, this approach does get closer to the Exaflops goal, but not until after 2020. It also, however, has a narrower window between the optimistic and pessimistic power models.

The performance per watt, Figure 7.15, is within one to two orders of magnitude by 2015, a

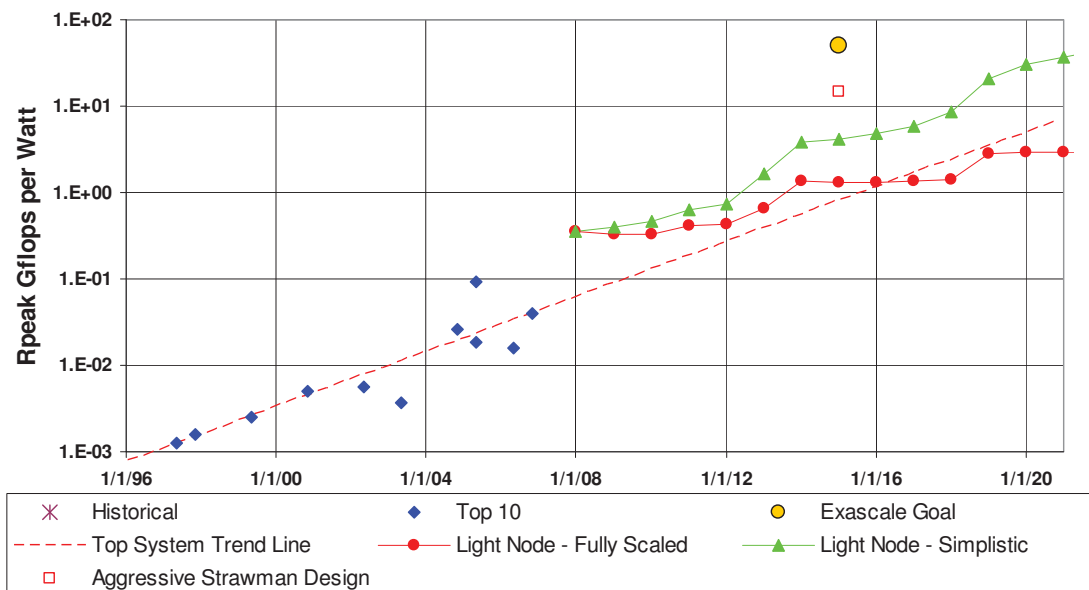


Figure 7.15: Light node strawman Gflops per watt.

substantial improvement over the heavy node strawman.

As much of an improvement this is, it still does not reach the Exascale goals. Looking closer into the details of the power model, a key observation that jumps out is that the power consumed in off-chip communication, and in memory references, is still a major contributor, and needs to be worked seriously.

7.3 Aggressive Silicon System Strawman

In this section we develop a strawman architecture for an Exascale computer system based on silicon, but with a clean sheet of paper and an aggressive but balanced approach to design options. While the design target is a data-center class system, the design will be done in a way that yields significant insight (discussed at the end) into what is feasible for both the departmental and embedded classes. The exercise indicates the scale of such a machine and exposes a number of design, programming, and technology challenges.

Our development proceeds in a bottom-up manner, starting with the floating-point units (**FPUs**) and working upward until we reach the level of a complete system.

Table 7.3 summarizes our bottom-up composition of a system. We assume a 2013 technology node of 32 nm as a baseline for the projection - this represents a reasonable technology out of which a 2015 machine might be fabricated. We start with an FPU (along with its register files and amortized instruction memory). Four FPUs along with instruction fetch and decode logic and an L1 data memory forms a **Core**. We combine 742 such cores on a 4.5Tflops, 150W active power (215W total) **processor chip**. This chip along with 16 DRAMs forms a **Node** with 16GB of memory capacity. The final three groupings correspond to the three levels of our interconnection network. 12 nodes plus routing support forms a **Group**, 32 Groups are packaged in a **rack**, and 583 **racks** are required to achieve a peak of 1 exaflops.

Level	What	Perf	Power	RAM
FPU	FPU, regs., Instruction-memory	1.5 Gflops	30mW	
Core	4FPUs, L1	6 Gflops	141mW	
Processor Chip	742 Cores, L2/L3, Interconnect	4.5 Tflops	214W	
Node	Processor Chip, DRAM	4.5Tflops	230W	16GB
Group	12 Processor Chips, routers	54Tflops	3.5KW	192GB
rack	32 Groups	1.7 Pflops	116KW	6.1 TB
System	583 racks	1 Eflops	67.7MW	3.6PB

Table 7.3: Summary characteristics of aggressively designed strawman architecture.

Year	Tech (nm)	V	Area (mm ²)	E/Op (pJ)	f (GHz)	Watts/Exaflops	Watts/FPU
2004	90	1.10	0.50	100	1.00	1.0E+08	0.10
2007	65	1.10	0.26	72	1.38	7.2E+07	0.10
2010	45	1.00	0.13	45	2.00	4.5E+07	0.09
2013	32	0.90	0.06	29	2.81	2.9E+07	0.08
2016	22	0.80	0.03	18	4.09	1.8E+07	0.07
2019	16	0.70	0.02	11	5.63	1.1E+07	0.06

Table 7.4: Expected area, power, and performance of FPUs with technology scaling.

Figure 7.16 diagrams the structure of one such group.

The strawman system is balanced by cost and our best estimate of requirements. The ratios of floating point performance to memory capacity in Table 7.3 are far from the conventional 1 byte per flops, or even from the “traditional” ratios found in the historical ratios from the Top 500 (Section 4.5.4) or in the strawmen of Sections 7.2.1 and 7.2.2. However it is roughly cost balanced, and with globally addressable memory across the system, should be adequate to hold at least some Exascale problems. It is, however, about 10X what 2008-2010 Petascale machines will have, implying that it is sufficient capacity for at least classes II and III of Section 5.6.1.

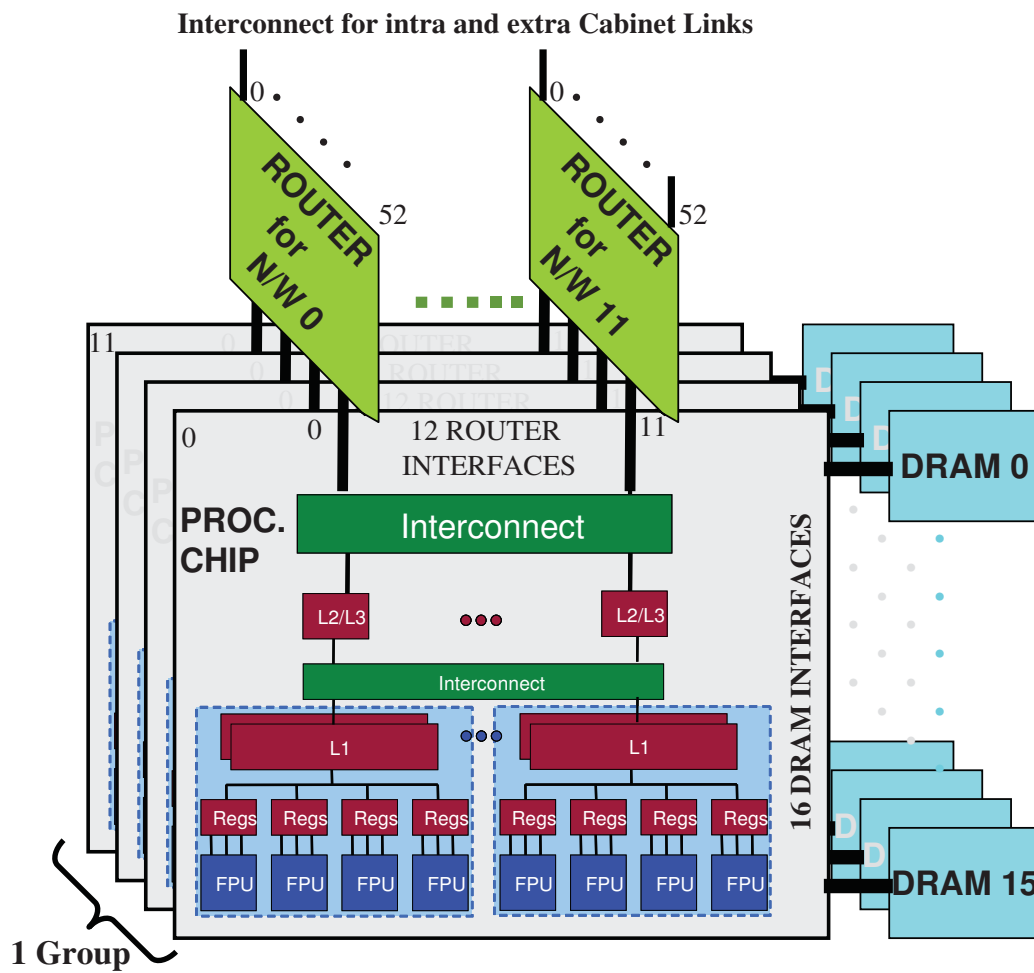
Similarly the bandwidth ratios of Table 7.7 do not provide a word of global bandwidth per flops, but rather budgets power across the levels of the storage hierarchy in a manner that returns the most performance per Watt. This is how implementable Exascale systems must be balanced.

In contrast to this fixed allocation, a system in which power allocation is adaptively balanced across levels of the bandwidth hierarchy is considered in Section 7.3.7.

7.3.1 FPUs

If we define an Exascale system as one capable of performing 10^{18} 64-bit floating-point operations, then at a minimum, our system needs a number of FPUs capable of this aggregate arithmetic rate. Table 7.4 shows how the area and power of FPUs is expected to scale with semiconductor technology. The first three columns of the table reflect the technology nodes projected by the International Technology Roadmap for Semiconductors (ITRS)[13]. The next three columns show how the area, energy per operation, and performance of a single floating point unit scale as line width (column 2) and voltage (column 3) are reduced. A **floating point operation (flop)** here is defined as a 64-bit floating-point fused multiply-add. The final two columns give the power to achieve an Exaflops (10^{18} 64-bit floating-point operations) with just FPUs (i.e. no overhead, memory, or transport), and the resulting power per FPU.

The baseline area, energy, and frequency for 90nm is taken from [42] and represents the power-



1 Cabinet Contains 32 Groups on 12 Networks

Figure 7.16: Aggressive strawman architecture.

Year	Tech (nm)	V	Area (mm ²)	E/Op (pJ)	f (GHz)	W/Exaflops	W/FPU
2004	90	0.8	0.50	52.9	0.6	5.3E+07	0.03
2007	65	0.8	0.26	38.2	0.9	3.8E+07	0.03
2010	45	0.8	0.26	38.2	0.9	3.8E+07	0.03
2013	32	0.6	0.06	10.6	1.5	1.1E+07	0.02
2016	22	0.5	0.03	5.1	1.9	5.1E+06	0.01
2019	16	0.5	0.02	3.7	3.1	3.7E+06	0.01

Table 7.5: Expected area, power, and performance of FPUs with more aggressive voltage scaling.

Unit	Energy per op (pJ)	Comment
FPU	10.6	Just the arithmetic operation
Register	5.5	Two reads, one write, 128 regs
Instruction RAM	3.6	32KB each access amortized across 4 FPUs
L1 Data RAM	3.6	64KB each access amortized across 4 FPUs
Core (per FPU)	23.3	Total core energy per FPU per op
Core (Total)	93.4	

Table 7.6: Energy breakdown for a four FPU processor core.

optimized FPU designed for the Merrimac Supercomputer. FPUs that are optimized for reduced latency or for higher operating frequencies may dissipate considerably more energy per operation. Energy scaling with process improvement is computed by assuming that energy per operation (CV^2) decreases linearly with line width (C) and quadratically with voltage (V^2). Maximum operating frequency is assumed to increase linearly as line width decreases.

The table shows that just the floating point units required to reach an Exaflops of performance will require 29MW in the 32nm technology node that will be available for a 2015 machine. Because this power number is prohibitively high, we also consider scaling supply voltage in a more aggressive manner than projected by the ITRS roadmap. The area and power of FPUs under this more aggressive voltage scaling is shown in Table 7.5. As above, energy is assumed to scale quadratically with voltage. Operating frequency is assumed to scale linearly with overdrive ($V_{DD} - V_T$) where we assume the threshold voltage V_T is 250mV.

Power lost to static leakage is not considered here but will be in the final numbers.

Table 7.5 shows that by aggressive voltage scaling we are able to significantly reduce the energy per operation (by nearly a factor of three at the 32nm node) at the expense of operating frequency. With this approach, the power required by the FPUs to achieve an Exaflops at the 32nm node is reduced to 11MW - just over 50% of the overall desired system power target.

Table 7.5 represents a fairly straightforward, if aggressive, voltage scaling of existing FPU designs. Further power savings may be possible by innovations in circuit design. Also, supply and threshold voltage can be optimized further — leading to even greater power savings at the expense of performance.

7.3.2 Single Processor Core

As a first step in building up to a processor core, we consider combining our FPU from the previous section with a set of local memory structures to keep it supplied with data, as summarized in Table 7.6. As we shall see, aside from the FPUs, the bulk of energy in our strawman machine will be consumed by data and instruction supply. At the 2013 32nm node, we estimate that a three-port

register file with 128 registers consumes 1.8pJ for each access and thus 5.5pJ for the three accesses (2 read and 1 write) required to support each floating-point operation. (Note a fused multiply-add actually requires a third read for a total of 7.3pJ). Adding in the FPU energy from Table 7.4 gives a total energy of 16.1pJ per operation for arithmetic and the first-level data supply.

Instruction supply and L1 data supply are dominated by memory energy. Reading a word from a 32KB memory array in this 32nm process requires about 15pJ. Hence, if we read an instruction from an I-cache for each operation, we would nearly double operation energy to 31.1pJ. However, by employing some degree of SIMD and/or VLIW control, we can amortize the cost of instruction supply across several FPUs. For our aggressive strawman design, we assume each instruction is amortized across four FPUs for a cost of 3.6pJ per FPU.

Similarly we assume that a single L1 Data Memory is shared by four FPUs for an energy cost of 3.6pJ per FPU. The energy of an L1 access is that to access a single 32KB L1 bank. However, the L1 memory may be composed of many banks to increase capacity. This can be done without increasing energy as long as only one bank is accessed at a time. For our aggressive strawman design, we assume that the L1 data memory is 2 32KB banks for a total of 64KB. For the purpose of the strawman we do not specify whether the L1 data memory is managed explicitly (like a scratch-pad memory that is simply directly addressed) or implicitly (like a cache which would require additional energy for tag access and comparisons). Most effective organizations will allow a hybrid of both approaches.

The energy breakdown of a 4-wide SIMD or VLIW processor core with registers, instruction RAM, and data RAM as described above is shown in Table 7.6. Note that even with sharing of the instruction and data memories across four FPUs the minimal functionality of a processor core more than doubles the energy required per operation to 23.3pJ. This raises the total power needed for an Exaflops machine to 23MW - again just considering the cores and their L1, and nothing above that.

This estimate has assumed an extremely power-efficient architecture. With a more conventional approach the energy per operation could be many times larger. Fetching one instruction per operation, employing complex control, permitting out-of-order issue, adding tag logic and storage, or increasing L1 data bandwidth, for example, could each increase the energy significantly. However, with careful design, the number suggested here should be approachable.

7.3.3 On-Chip Accesses

At the next level of our strawman architecture, we consider combining a number of our simple 4-way cores on a single chip. The number of cores we put on a single chip is limited by power, not area. In this power-limited regime the architecture of our multi-core chip is directly determined by its power budget. We assume a limit of 150W for the active logic on chip and consider the budget shown in Table 7.7. Given a power allocation of 70% (105W) for the cores (59% FPUs + 11% L1) the chip can support 742 4-way cores (2968 FPUs) for a peak aggregate performance of 4.5 teraflops. As shown in the table, the amount of power allocated to each level of the storage hierarchy determines the bandwidth (BW in GWords/s) at that level. The column labeled "Taper" is the number of FPU operations per single word access at each level. The numbers in the DRAM and Network rows reflect only the energy consumed in the multi-core processor die. Accesses at these levels consume additional energy in memory chips, router chips, and transceivers.

Note that to improve the yield of these ~3K FPU chips some number of additional cores and memory arrays would be fabricated on the chip as spares that can be substituted for bad cores and memory arrays during the test process via configuration. Alternatively one can simply configure all "good" cores on each chip for use and allow the number of cores per chip to vary from chip to

Item	Percent	Watts	Units	BW (GW/s)	Taper	Comments
FPU's	59.0%	88.5	2968	4495	1	Includes 3-port reg and I-mem
L1 Data	10.9%	16.4	742	1124	4	64KB per 4 FPU's
L2	6.9%	10.4	371	562	8	256KB per 2 L1s
L3	7.5%	11.3	189	286	16	Global access to L2s
DRAM	10.0%	15.0	59	89	50	Attached to this chip
Network	5.6%	8.4	13	27	164	Global access
Taper = number of flops executed per access						

Table 7.7: Power budget for strawman multi-core chip.

chip.

At some distance, the energy required to access on-chip memory becomes dominated by the energy required to communicate to and from the memory. The distance at which this occurs depends on the signaling technology used. At the 32nm process node we estimate on-chip line capacitance at 300fF/mm. With an 0.6V supply and full-swing signaling this gives signaling energy of 110fJ/bit-mm, or 6.9pJ/word-mm for a 64-bit word. With a signaling protocol that uses a reduced signal swing of 0.1V, these numbers can be reduced to 18fJ/bit-mm and 1.2pJ/word-mm. With array access energy of 14.6pJ, the point at which access via full-swing signaling becomes dominated by signaling energy is at a distance of 2.1mm. With more efficient 0.1V signaling, this crossover distance is increased to 12.7mm. For the rest of this analysis we assume the use of the more efficient signaling system with 0.1V signal swings.

For our strawman design, as pictured in Figure 7.16 global on-chip memory consists of a 256KB RAM (composed of 32KB subarrays) for every two 4-way cores, for a total of 371 arrays for 92.8MB on-chip storage (again we ignore the distinction between cache and directly addressable memory). These arrays are connected to the processor cores using an on-chip interconnection network that uses efficient signaling. To account for locality we divide all of this on-chip memory into "L2" and "L3" levels. Both of these levels share the same on-chip memory arrays. L2 accesses reference memory arrays associated with a local group of 16 4-way cores. L3 accesses are to any location on the chip. Each L2 access requires traversing an average of 3.4mm of network channels while an L3 access requires an average of distance of 21.3mm. Note that the memory arrays can support much higher bandwidth than is provisioned at all three levels. However the power budget (and possibly the on-chip network bandwidth) limits the bandwidth to the level shown in the table. The numbers in the table consider only the energy needed to transport payload data over the on-chip interconnect. This is slightly optimistic; control overhead will add to the energy at these levels - the amount of overhead depends on the granularity of transfers. Our data-only estimates are accurate enough for the purposes of this strawman.

The DRAM and Network rows of Table 7.7 assume an energy of 2pJ/bit to cross between two chips. The DRAM number includes one global communication on chip (21.3mm) and one chip crossing. The network number includes one global on-chip communication and two chip crossings - one to get to the router, and a second to get to the DRAM on the destination chip. Additional energy consumed in the DRAM chips and the network are discussed below.

Table 7.8 shows the area breakdown for the multi-core processor chip. Power limits FPU's (which includes register files and instruction memories) to occupy less than half the chip. The capacity of the L1-L3 arrays is then adjusted to fill the remaining area on the die. Note that the capacity of the memory arrays sets their area while the bandwidth provided at each level sets their power allowing these two parameters to be adjusted independently.

Item	Units	Each (mm ²)	Total (mm ²)
FPU's	2968	0.06	188
L1	742	0.09	66
L2/L3	371	0.36	132
			386

Table 7.8: Area breakdown of processor chip.

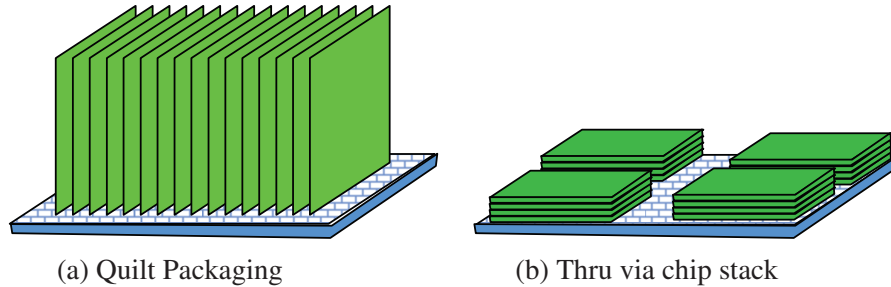


Figure 7.17: Possible aggressive strawman packaging of a single node.

Finally, as discussed in Section 6.2.2.1, leakage currents will still be significant, and account for roughly 30% of total chip power. This would raise the actual processor chip power to about 215W.

7.3.4 Processing Node

As pictured in Figure 7.16, a **processing node** consists of the multi-core processor chip described above and 16 1GB DRAM chips for an aggregate memory of 16GB/node. Each DRAM chip is assumed to have a 32-bit wide interface that operates at 11Gb/s per signal for a per-chip bandwidth of 5.5GWords/s. The 16 chips provide an aggregate bandwidth of 88GWords/s - slightly less than the 89GWords/s specified in Table 7.7. This DRAM interface consumes 512 differential signals (32 per DRAM chip) on the processor chip for the data path. We assume the control/address path consumes an additional 128 differential signals (8 dedicated to each DRAM), for a total of 640 signals.

Figure 7.17 diagrams two ways in which such a node may be packaged so that chip to chip interconnect is as direct as possible.

A contemporary commodity DRAM chip has an access energy of about 60pJ/bit. However, analysis suggests that by 2013 a DRAM can be built that can access an internal array with an energy of 0.5pJ/bit, transport a bit from the sense amp to the interface circuitry with an energy of 0.5pJ/bit, and drive a bit off chip with an energy of 2pJ/bit for a total access energy of 3pJ/bit. We use this more aggressive DRAM energy estimate here to assume design of a specialized DRAM optimized for high end systems. The internal access energy includes charging a 100fF bit line to the full power supply (0.1pJ) and 5× overhead. The transport includes driving 12mm of wire through a low swing and associated drivers, receivers, and repeaters.

A total energy of 3pJ/bit (or 192pJ/word) and a frequency of 5.5GW/s gives a power of 1.1W per chip or 17W for the aggregate node memory. This brings the total node power to 230W. A breakdown of how this power is distributed is shown in Figure 7.18.

The node memory could be stacked horizontally or vertically on the multi-core processor die.

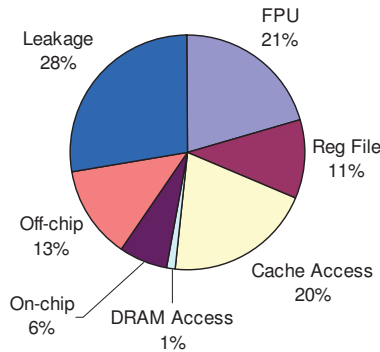


Figure 7.18: Power distribution within a node.

However this would only be useful to the extent that it reduces the energy per unit bandwidth for accessing the DRAM or the cost of providing the high pin count otherwise required. The numbers described in this strawman can be realized using existing signaling technology packaged on conventional printed circuit boards.

By conventional measures having 16GB of memory associated with a 4 teraflops processor seems small. However, this ratio is roughly balanced by cost and by interface bandwidth. Modern DRAMs are connected point-to-point and creating larger memory systems by aggregating more memory chips requires interface chips to terminate the signal pins on these memories. In this strawman design, the processor chips act as these interface chips, but perform arithmetic operations as well.

Also, within this estimate we do not assume any leakage power on the DRAMs - in actuality there may be some due to the enhanced logic circuits assumed, but they would be dwarfed by the processor chip.

Note that an Exaflops system requires 223,000 processing nodes and provides 3.4PB total memory.

7.3.5 Rack and System

Like the processor die, the contents of a rack are power limited. We assume a single rack can support 120kW. The amortized power of interconnection network components is 62.7W/node, bringing the total node power to 295W/node. Hence we can support up to 406 processing nodes in one rack. To make the network a bit simpler we will use $12 \times 32 = 384$ nodes (283,444 cores and 1,133,776 FPUs) per rack. The aggregate floating point performance of the rack is 1.7 petaflops. A total of 583 racks is required to provide an exaflop of performance.

7.3.5.1 System Interconnect Topology

A simple interconnection network, based on a **dragonfly topology** is used to connect the processors in a rack. This network is seamlessly connected to the on-chip network and provides a global memory address space across the entire system. To provide the aggregate global node bandwidth of 20GWords/s (1260Gb/s) in Table 7.7 we provide each processor chip with 12 full duplex 4-bit wide **channels** that operate with a 30Gb/s signaling rate. Each of these 12 channels connects to a separate parallel network and traffic originating at the node is load balanced across them. Global bandwidth can be smoothly varied by provisioning fewer than all 12 parallel networks. Figure 7.19 diagrams these connections.

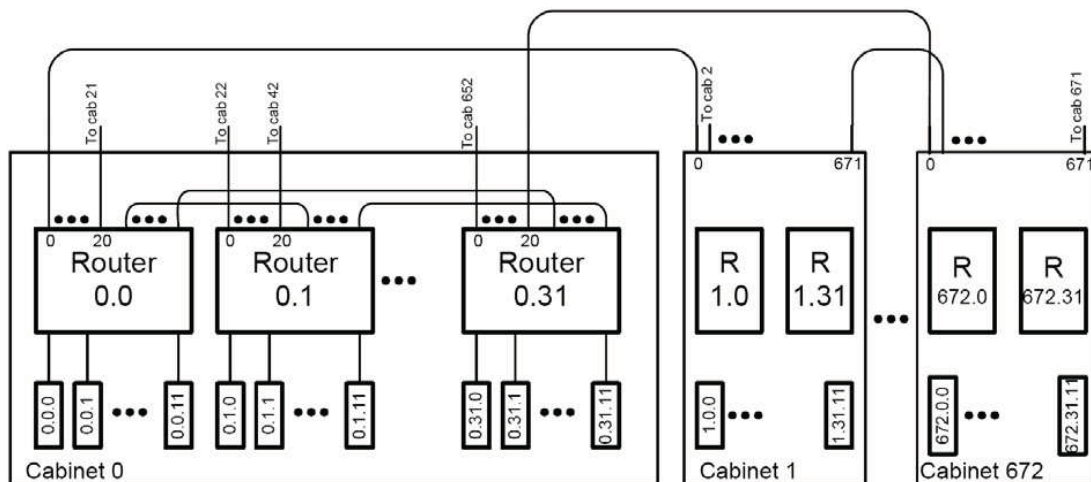


Figure 7.19: The top level of a dragonfly system interconnect.

Each of the 12 parallel networks is constructed by dividing the 384 nodes in each rack into 32 **groups** of 12 processors each. A radix-64 **router chip** is associated with each of the groups giving 32 routers for each of the 12 parallel networks, or a total of 384 router chips (the same number as the processor chips). The 61 channels on each router chip are divided into 12 **processor channels** (one for each processor in the group), 31 **local channels** (one for each other group), and 21 **global channels** (each of which goes to a different rack). Each group's router connects to a different set of 21 racks so that the $21 \times 32 = 672$ global channels out of each rack reach 672 different racks enabling machines of up to 673 racks. For smaller configurations multiple connections are made to the same distant rack in a modulo manner.

The network can be routed minimally or non-minimally. For a **minimal route** to a distant rack, a packet traverses between two and four routers:

1. It is forwarded to any of the 12 routers connected to its source processor (the source router). This picks one of the 12 parallel networks.
2. It is forwarded over a local link to a router within the current parallel network with a connection to the destination rack, unless it is already at that router or is already in the correct rack. This selects the group within the source rack attached to the correct destination rack.
3. It is forwarded over a global link to the destination rack unless it is already at the correct rack. This selects the destination rack.
4. It is forwarded over a local link to a router connected to the destination processor unless it is already there. This selects the destination group.
5. Finally, the packet is forwarded to the destination processor for further forwarding over the on-chip interconnection network within this processor chip.

Note that each minimal packet traverses two local links and one global link. Hence with our ratio of 12:32:21 processor:local:global links the processor links will be the bandwidth bottleneck as is desired to deliver the specified node bandwidth.

Like all butterfly derivatives, the dragonfly can become unbalanced using minimal routing on adversarial traffic patterns. In such cases **non-minimal routing** must be employed - for example by using global adaptive load-balanced routing. This is accomplished by forwarding the packet to a different intermediate rack (randomly or adaptively selected) before forwarding it to the destination. In this case, a route traverses up to three local and two global channels. For such non-minimal routes the global channels become the bottleneck. However the network is very close to balanced.

7.3.5.2 Router Chips

Our radix-64 router chip has 64 full duplex channels, with each path consisting of a 4-signal high-bandwidth differential link, all operating at 30Gb/s. For purposes of estimating energy, we assume that transmitting each bit across a router takes 4pJ - 2pJ for the energy to drive the outgoing link, and 2pJ to traverse the router internals. This gives an overall router active power of 30.7W. Applying the leakage tax discussed earlier to the on-chip portion raises this to about 37.5W. We also assume that the 21 global channels attached to each router require an additional 10pJ/bit for external transceivers or buffers, so the 2520Gb/s of global channel bandwidth out of each router requires an additional 25.2W giving a total interconnect power of 62.7W per node. Note that global bandwidth can be varied independently of local bandwidth by omitting global channels from some of the parallel networks. This allows a system to be configured with a desired amount of local bandwidth - by provisioning N (between 1 and 12) parallel networks, and separately setting the amount of global bandwidth by provisioning G (between 1 and N) of these networks with global channels.

7.3.5.3 Packaging within a rack

While there are many options as to how to package all the logic that goes into one rack, we assume here that each group of 12 processor chips, their 192 DRAM chips, and the associated 12 router chips are all packaged on a single board, with edge connections for both the local and global channels. There are up to $52 \times 12 = 624$ such channels that must be connected to each such board. 32 of these boards make up a rack, along with the disks, power supplies, and cooling subsystems.

We may also assume that each of these 32 boards plug into a single backpanel, eliminating the need for cabling for the channels connecting groups within a rack. This accounts for all but 21 of the channels of each of the 12 routers per group. The remaining $12 \times 21 = 252$ channels per board (8064 overall) are distributed to the other racks as pictured in Figure 7.19. Since there are 12 parallel networks, it makes sense to assume that the 12 channels that interconnect each rack are “bundled” into a single cable, meaning at least 582 cables leave each rack - one to each of the other racks.

7.3.6 Secondary Storage

To provide checkpoint storage, scratch space, and archival file storage up to 16 disk drives are associated with each group of 12 processors. These drives are located in the rack and attached via high-speed serial channels (the evolution of SATA) to the routers in the group. They are accessible via the network from any processor in the system. Using projections for 2014 disk drives (Section 6.4.1), the 16 drives will provide an aggregate of 192TB of storage, 64GB/s of bandwidth, and dissipate about 150W. The 64GB/s of bandwidth is sufficient to checkpoint the 192GB of DRAM storage in the group in 3 seconds, if they can all be run in parallel at full bandwidth.

Assuming a node MTBF of 10^6 hours we have a system MTBF of about 3 hours. With a checkpoint time of 3 seconds (8.3×10^{-4} hours), a checkpointing interval of about 3 minutes is near

Item	Percentage	Watts	Units	Bandwidth	Taper
FPU's	84.3%	126.5	4240	6421	1
L1	62.5%	93.7	4240	6421	1
L2	79.1%	118.6	4240	6421	1
L3	99.7%	149.6	2523	3821	1.7
DRAM	100.0%	150.0	592	897	7
Network	100.0%	150.0	234	354	18

Table 7.9: Power allocation for adaptive node.

optimal which gives a checkpoint overhead of about 1.7% (see Section 6.7.4).

For scratch and archival storage, the disks provide an aggregate of 6.1PB of storage for each rack and 3.6EB of storage in an Exaflops system. Larger amounts of storage if needed can be provided external to the main system. Given that the configuration supports a total of 3.4PB of DRAM memory, this 3.6EB of disk storage should be more than sufficient for both checkpointing, and scratch usage, even with RAID overheads for redundancy. In fact, it may very well offer at least rudimentary archival support.

An intermediate level of non-volatile memory - with latency, bandwidth, and cost/bit between DRAM and disk would be valuable to reduce the latency of the checkpoint operation and speed access to frequently accessed files. For example, suppose a technology existed for memory chips with 16GB density (16x the density of the 1GB DRAMs expected in 2014) and a

bandwidth of 5.5GB/s (1/8 the bandwidth of the DRAM chips). An array of 8 of these chips associated with each processing node would provide ample storage for checkpointing and would provide an aggregate bandwidth of 44GB/s - reducing the time to take a checkpoint to 0.36s which is nearly an order of magnitude faster than checkpointing with disks.

7.3.7 An Adaptively Balanced Node

In Sections 7.3.2-7.3.5 we architected a strawman system with a fixed allocation of power to the various levels of the hierarchy. In this section, we revisit this design and sketch a system in which *each* level of the hierarchy is provisioned so that it can consume *all* of the power allocation by itself. The result is a node organization with higher bandwidth at each level of the hierarchy than shown in Table 7.7. To prevent a chip from oversubscribing the total power limit, a throttling mechanism is used to monitor overall power use (e.g. by counting the number of accesses at each level of the hierarchy) and throttle instruction issue or other performance mechanisms when the short-term average power exceeds a threshold.

With this *power-adaptive* organization all levels of the hierarchy cannot operate a maximum capacity simultaneously — or the overall power limit would be exceeded by many fold. However, this arrangement enables each node to *adapt* its power use to the application at hand. An application segment that demands very high DRAM bandwidth and relatively little floating-point can use all of its power on DRAM bandwidth. A different application segment that can operate entirely out of registers may use all of its available power on floating point, with little for memory or communication. Together, this may allow a single design to match more readily to a wider class of applications, even though for any one class some part of the machine's resources are "under-utilized."

Table 7.9 shows the power allocation for the adaptive node. We provision 4240 FPUs in 1060 cores. At our 1.5GHz clock rate, this provides a peak performance of 6.4Tflops when operating out of registers. When operating at peak rate from registers, 84.3% of the power is consumed

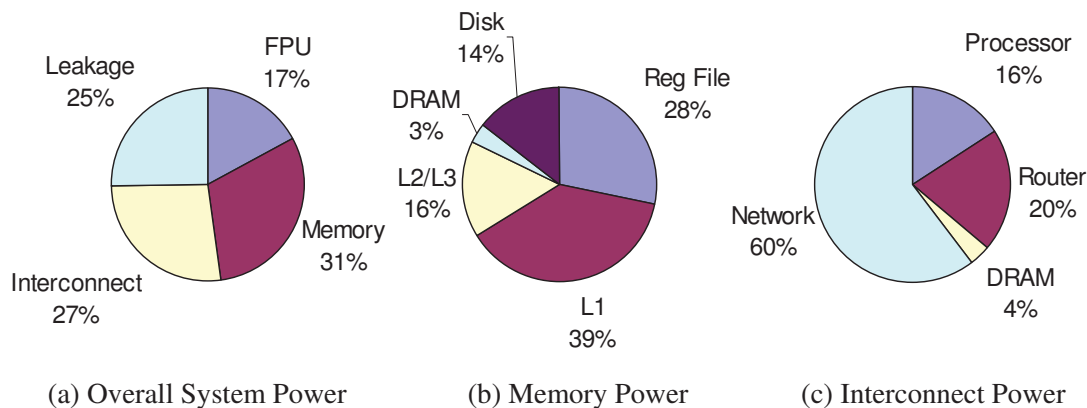


Figure 7.20: Power distribution in aggressive strawman system.

by the FPUs with the balance consumed by accessing registers, fetching instructions, and control overhead.

For on-chip access, we provision the L1 and L2 so they can each support one access for every floating-point unit each cycle. This requires 62.5% and 70.1% of the total power respectively. Thus, we cannot maintain the peak floating-point rate at the same time that these memories are being accessed at full bandwidth. To stay within the power envelope, we restrict L3 accesses to one access for each 1.68 floating point units per cycle. This consumes all available power.

For off-chip access, both the DRAM interface and the network interface are also configured so that they can each consume all available power. This widens the DRAM interface by an order of magnitude to a bandwidth of nearly 900GWords/s. The network bandwidth is half of this rate. With this adaptive power provisioning, the off-chip data rates required now pose packaging challenges where with the fixed budget shown above they were well within the capabilities of existing technologies.

7.3.8 Overall Analysis

Figure 7.20(a) breaks down the power usage within the aggressive strawman design. The categories include that for the FPUs alone, for accessing any memory structure in the system, for interconnect (both on and off-chip), and for leakage. As can be seen, power is fairly evenly distributed, with FPU power the lowest and memory power the highest.

Figure 7.20(b) then breaks down just the power associated with accessing any memory structure in the system. This distribution is much less evenly divided, with about 2/3 of the power spent closest to the FPUs in the register files and in the L1 instruction and data caches. The L1 energy is evenly split between instruction and data accesses.

Figure 7.20(c) diagrams the distribution in power spent in transferring data, both on-chip and between chips. Here the clear bulk of the power is spent in transferring data between racks.

7.3.9 Other Considerations

The power estimates above are a clear “best-case” scenario - it is very highly unlikely that any real implementation is likely to come in at any lower power levels. The estimates above do consider the effects of leakage current - at the system level it raises total power from 49.8MW to 67.7MW, or

an overall tax of about 26%. However, there are several other factors that may in fact increase the total power of a real system, but are not in the above estimate:

- The ratio of flops (1 exaflops) to memory (3.6PB) is 0.0036 - about two orders of magnitude less than what is seen in many of today's supercomputers, such as described in the heavy evolutionary strawman of Section 7.2.1.
- There is no ECC on the memory - this might apply about a 12% upside to both the memory and the transfers between memory and processor chips.
- There is no energy expenditure for matching tag arrays in caches - we treat the on-processor arrays as directly addressed "memories" in order to get a minimum energy measure.
- Energy for data references from any memory is for the exact 8-byte words desired - no extra transfers for line prefetch is accounted for.
- Energy for address and command transfers on the processor-memory interfaces is not considered.
- Likewise, ECC and protocol overhead needs to be considered on the node-node and group-group links.
- There is no overhead for clock distribution on the processor chips - in many real chips today this is a significant percentage of the total die power.
- There is no overhead for redundancy management in the processors, such as duplicate cores, spare FPUs, comparators, etc.
- There is no overhead for redundancy and RAID controllers in the scratch disk estimates.
- Other than the register files, there is no energy expenditures accounting for core instruction decode logic, or for non-FPU processing such as address computations.

7.3.10 Summary and Translation to Other Exascale System Classes

Although the Exascale system described above targeted the data center class system, pieces of it can be used to get a handle on the characteristics of the other two classes: departmental and embedded. To drive such a discussion, Table 7.10 summarizes estimates for five different variations:

1. **Exaflops Data Center:** represents the aggressive design as discussed earlier - a system that has a peak capability of 1 exaflops regardless of power or other limits.
2. **20 MW Data Center:** the same system derated to fit a 20 MW power limit. This yields perhaps 30% of an exaflops, and thus represents the best we can do with silicon in a 20 MW envelop.
3. **Departmental:** A single rack from the first column.
4. **Embedded A:** A single processor die and its 16 DRAM chips. Neither a router chip nor disk drives are counted.
5. **Embedded B:** the same as Embedded A, but where the number of cores (and thus power) on the processor die has been derated to a peak of one teraflops.

Characteristic	Exascale System Class				
	Exaflops Data Cen- ter	20 MW Data Cen- ter	Department	Embedded A	Embedded B
Top-Level Attributes					
Peak Flops (PF)	9.97E+02	303	1.71E+00	4.45E-03	1.08E-03
Cache Storage (GB)	3.72E+04	11,297	6.38E+01	1.66E-01	4.03E-02
DRAM Storage (PB)	3.58E+00	1	6.14E-03	1.60E-05	1.60E-05
Disk Storage (PB)	3.58E+03	1,087	6.14E+00	0.00E+00	0.00E+00
Total Power (KW)	6.77E+04	20,079	116.06	0.290	0.153
Normalized Attributes					
GFlops/watt	14.73	14.73	14.73	15.37	7.07
Bytes/Flop	3.59E-03	3.59E-03	3.59E-03	3.59E-03	1.48E-02
Disk Bytes/DRAM Bytes	1.00E+03	1.00E+03	1.00E+03	0	0
Total Concurrency (Ops/ Cycle)	6.64E+08	2.02E+08	1.14E+06	2968	720
Component Count					
Cores	1.66E+08	50,432,256	2.85E+05	742	180
Microprocessor Chips	223,872	67,968	384	1	1
Router Chips	223,872	67,968	384	0	0
DRAM Chips	3,581,952	1,087,488	6,144	16	16
Total Chips	4,029,696	1,223,424	6,912	17	17
Total Disk Drives	298,496	90,624	512	0	0
Total Nodes	223,872	67,968	384	1	1
Total Groups	18,656	5,664	32	0	0
Total racks	583	177	1	0	0
Connections					
Chip Signal Contacts	8.45E+08	2.57E+08	1.45E+06	2,752	2,752
Board connections	1.86E+08	5.65E+07	3.19E+05	0	0
Inter-rack Channels	2.35E+06	7.14E+05	8,064	0	0

Table 7.10: Exascale class system characteristics derived from aggressive design.

For each of these system classes there are four sets of summary characteristics:

1. Some key absolute functional attributes from Section 2.1, namely peak performance, total storage, and power.
2. Some normalized attributes.
3. Component counts.
4. Counts of connections of various sorts: chip, board edge, and channels between boards. The chip contacts are individual contacts such as C4 pads that support individual signals - power, ground, clocking, and control contacts are not counted. The board edge counts assume a board with 12 nodes (processor + router + DRAMs) on it, and again represents individual connector "pins" that carry individual electrical signals (two needed for a differential link). The inter-rack channels represent the number of distinct router compatible channel paths that couple racks.

The last two sets of metrics were included to give some sense of how reliability issues may come into play, especially for larger systems.

7.3.10.1 Summary: Embedded

The two embedded columns indicated that even though a processor chip plus DRAMs make a rather potent assembly of about the performance desired for an embedded class implementation, the power of at least 153 W is still excessive by some non-trivial factor, even considering the low 0.014 bytes per flop present in the 1 teraflops configuration. Boosting this memory capacity to something more reasonable will increase power in two ways: per DRAM chip power and increased power per processor-memory link. The latter is likely because there are insufficient contacts available in silicon at that time to allow for many more memory interfaces, meaning that each interface must now support multiple DRAM chips and the added capacitance they introduce.

In terms of memory bandwidth, the Embedded B column does support more bandwidth per flop, mainly because we assume the number of cores drops to 180 from 742. However, no additional bandwidth is gained by adding more DRAMs.

Also, there is still considerable concurrency - at least 720 flops per cycle must be extracted from the application in order to achieve peak efficiency. This is orders of magnitude above any "embedded class" processing system of today.

7.3.10.2 Summary: Departmental

The Departmental system consists of one rack from the exaflops system. This is again in the right performance range of 1.7 petaflops (if we can run at 100% efficiency) and is the right physical size (1 rack), but is again significantly off the mark in power and memory capacity. While the active power of the electronics for a single rack is 116 KW, the overhead for power supply inefficiencies and cooling may raise the wall-plug power to upwards of 200KW - probably too much for a small machine room. Also 0.0036 bytes per flop is 10-100X below the expected Petascale machines of 2010, of either the heavy or light node types of Section 4.5. With the percentage of power due to memory at 29% (Figure 7.20), getting to even 0.1PB of DRAM would add on the order of 1/2 MW to the total.

Concurrency is also considerably higher - on the order of 1 million operations must be independently specified by the program in each and every machine cycle.

Component	FITs/Component	Exascale 1		Exascale 2	
		# Components	FITs	# Components	FITs
Processor chip	1000	224K	224M	224K	224M
DRAM chip	5	3,582K	18M	14,330K	72M
Flash chip	5	1,791K	9M	7,164K	36M
Router chip	1000	224K	224M	224K	224M
Disk Drive	1000	229K	299M	299K	299M
Power Supply	100	37K	4M	37K	4M
HW FITs			777M		857M
Other FITs			777M		857M
Total FITs			1,554M		1,715M
MTTI (minutes)			39		35

Table 7.11: Failure rates for the strawman Exascale system.

Also, from the packaging perspective, the number of individual pins carrying individual signals between each board and the backplane is on the order of 5,000 pairs (52x12x4x2), which is not practical with any technology of today.

On the bright side, for the amount of DRAM capacity in this system the disk capacity is 1000X - more than ample for scratch or secondary, and bordering on useful for a more general file system with some archival properties.

7.3.10.3 Summary: Data Center

The same comments spoken for the departmental system can be applied to the Data Center class. As discussed in Chapter 5, different applications may scale from today to an exaflops in several different ways, with a weather model, for example, taking anywhere between 10 and 100 PB of main memory (Section 5.7.2). Even the lower of these numbers is almost 3X the assumed capacity, meaning that the 67MW number would be low by an additional large factor. Again we have not factored in environmental power considerations.

Concurrency in the full exaflops system nears one billion operations in each cycle.

Some additional significant reliability concerns also surface in this class. There are over 4 million chips, three hundred thousand drives, almost a billion signal-carrying chip contacts, and about 170,000 cable bundles between racks that are each carrying about a dozen channels. Tripling the memory to a mere 10PB adds another 10 million chips and 1 billion chip contacts.

7.4 Exascale Resiliency

We can now analyze the resiliency of the aggressive silicon strawman described in Section 7.3. Table 7.11 shows an overview of two sample Exascale Systems: Exascale 1 with 16GB of DRAM per processor chip and Exascale 2 with 64GB of DRAM per processor chip. According to the strawman, we assume 16 disk drives per 12-node cluster and 8–32 flash chips per processor chip, for data storage and checkpointing. We assume the same failure rate for Flash and DRAM (5 FITs per chip) as was budgeted for BlueGene/L assuming that the decrease in single event upset rates (SEUs or soft errors) will be offset by a slow increase in hard failures. The processor and router chips are custom low-power circuits and we assume 1000 FIT per chip based on SIA projections, which accounts for improvements in chip-level resiliency as well as more severe chip failure modes.

		Exascale 1	Exascale 2
Disk Checkpoints (sustained 270 MB/sec)	Checkpoint Latency (Seconds)	60.0	240.0
	Availability	77%	52%
Disk Checkpoints (sustained 2.7 GB/sec)	Checkpoint Latency (Seconds)	6.0	24.0
	Availability	93%	85%
Flash Checkpoints (sustained 22 GB/sec)	Checkpoint Latency (Seconds)	0.7	2.9
	Availability	97%	95%

Table 7.12: Checkpointing overheads.

We assume a 100,000 hour reliability on disk drives (1000 FIT) and a lower 100 FIT rate on power supplies.

These assumptions produce a hardware failure rate of 777–857 million FIT for an Exascale system. Based on experience that hardware accounts for about half of the overall system failures, an Exascale system could be expected to have a failure rate of 1.5–1.7 billion FITs. These failure rates corresponds to a failure every 35–39 minutes.

Table 7.12 provides an analysis of the effect of failure rate on the availability of the system. For this analysis, availability is defined to be the fraction of time that the machine operates at full capacity, assuming that repair time is zero. While aggressive sparing with automatic fail-over can help a system approach ideal zero-repair time, the system will still experience some degradation. The table shows checkpoint latency for three different scenarios: (1) local disk checkpointing at 270 MB/second (1/20 the raw disk bandwidth) to account for system overheads, (2) local disk checkpointing at 2.7 GB/second (optimistically 1/2 the raw disk bandwidth), and (3) local flash checkpointing at 22 GB/second (1/16 of the raw DRAM bandwidth). The total checkpointing latency is a function of both the checkpointing bandwidth and the memory capacity, assuming checkpointing of the entire contents of DRAM. The slowest checkpointing rates are 1–4 minutes, while the fastest are 1-3 seconds. The machine utilization is computed using an optimal checkpointing interval that maximizes the availability, accounting for the overhead to take a checkpoint as well as the work lost when a failure occurs. The utilization ranges from 52% to 95% depending on the checkpointing bandwidth, failure rate, and memory capacity.

This analysis emphasizes the importance of design and technology developments on the capabilities of the machine, as utilization degradation requires a larger machine to sustain a given level of performance. Even 90% utilization requires a machine to have 10% more components and consume 10% more power to achieve a particular Exascale metric. This analysis also indicates that fast rollback and recovery schemes, coupled with automatic fail-over, can reduce the effect of significant failure rates. If these mechanisms are sufficiently effective, even higher failure rates can be tolerated, which gives designers the opportunity to choose less reliable components or technologies as a part of the cost/power/performance/utility optimization.

7.5 Optical Interconnection Networks for Exascale Systems

We develop here an exercise to explore a simple model for the insertion of photonic interconnect technologies within the context of the strawman Exascale system design in 7.3. The goal of this analysis is to expose key system metrics which may be enabled by optical interconnects, particularly potential gains in the available bandwidth under an equivalent power budget to the strawman design. The unique challenges associated with optical technologies (i.e. the lack of equivalent optical RAM) will of course require new architectural approaches for the interconnection networks.

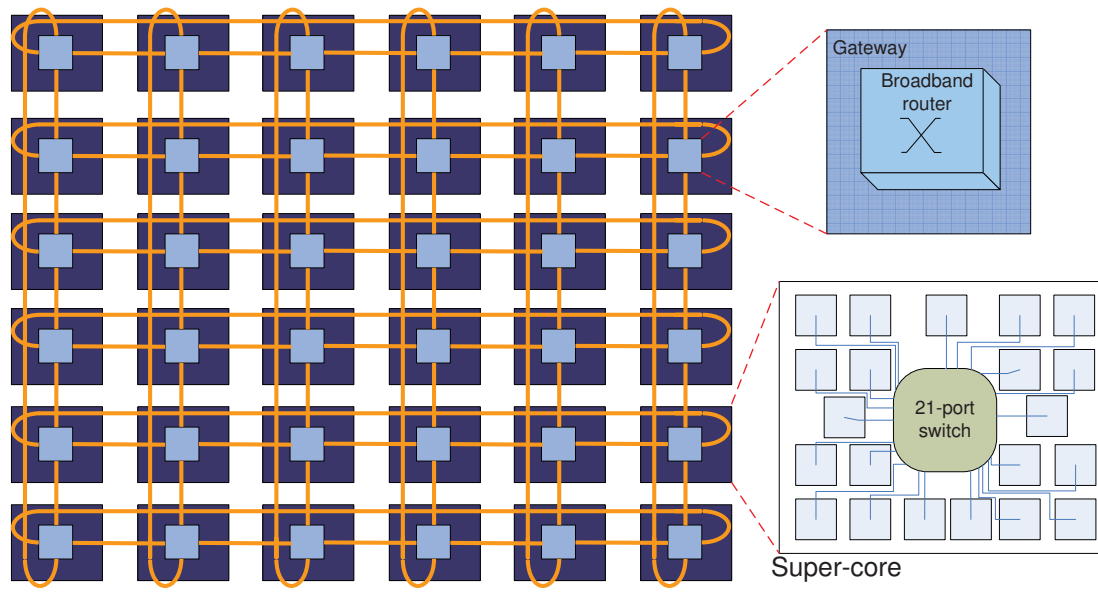


Figure 7.21: Chip super-core organization and photonic interconnect.

The analysis here however makes no attempt to design any aspects of the interconnection network and makes only broad assumptions regarding the topology, flow control, etc.

7.5.1 On-Chip Optical Interconnect

We begin as in the silicon system strawman design (7.3) in a bottom-up fashion with the processor chip. We organize the processor chip into groups of cores, or **super-cores**. For the 742-core chip, we have 36 super-cores each containing 21 cores (see Figure 7.21). An optical Gateway which contains the electronic/photonic plane interface as well as a routing switch is associated with each super-core. The super-cores form a regular 6x6 grid on-chip which is connected via a photonic on-chip network. The gateways provide the electro-optic (E/O) and opto-electronic (O/E) interfacing between the processor super-cores and the photonic interconnect. The optical Network on-chip (NoC) network topology can be assumed to be a mesh, or some derivative thereof, such as a torus. In addition to the E/O and O/E interface, the gateways include a broadband router switch which can direct a multi-wavelength optical message either onto the on-chip network or to an off-chip fiber port. The gateway router switch can be configured to receive multi-wavelength messages from off-chip. The on-chip electronic network as described in the strawman is assumed to remain and its full power budget of 8.4W will be included in the calculations. However as the design is further optimized and the optical on-chip network is used for an increasing fraction of the traffic (particularly for large, high-bandwidth message transfers) the power budget allocated to the on-chip electronic interconnect may be reduced.

7.5.2 Off-chip Optical Interconnect

The strawman design chip is rated at 4.5 Tflops. We assume here that the optical NoC can provide 1 B/s for each flop. This is equivalent to 36 Tb/s available to the chip or 1Tb/s per super-core. In addition, the 36 broadband router switches at each gateway can direct 1Tb/s to an off-chip fiber

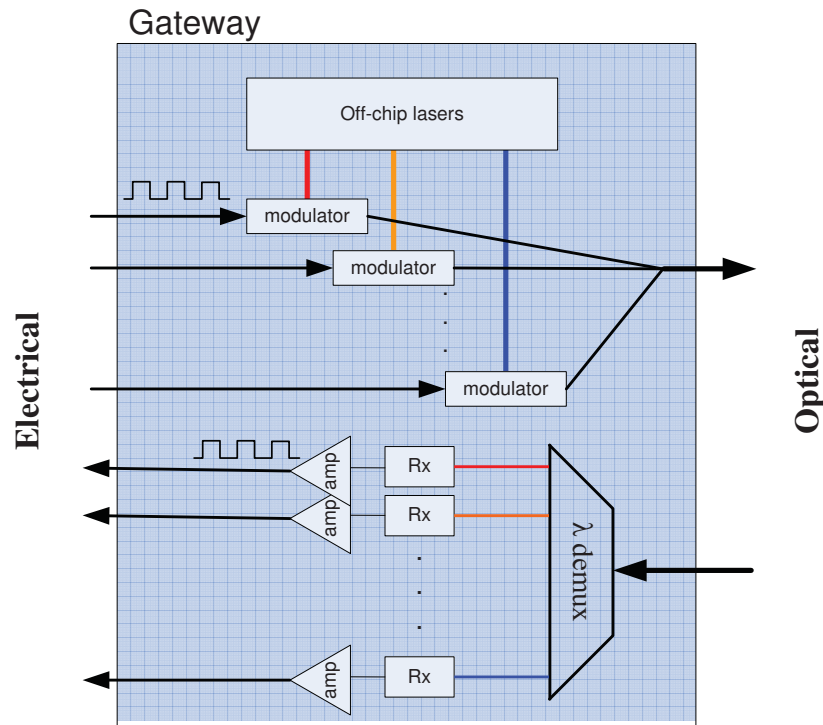


Figure 7.22: Gateway functional block design.

Modulator power	0.1 pJ/bit
Receiver power	0.1 pJ/bit
On-chip BroadBand Router Power	0.5 mW
Power per laser	10 mW
Number of wavelength channels	250

Table 7.13: Optical interconnect power parameters.

port and are configured to receive 1Tb/s from an off-chip port (these broadband ingress/egress switches are not explicitly shown in the figure).

The gateway (see Figure 7.22) E/O interface consists of a bank of electro-optic modulators fed by multiple off-chip lasers (one for each of the wavelengths) and the electronic signaling from the processor plane. The O/E interface consists of passive wavelength demultiplexing and a bank of receivers (one for each of the wavelengths).

Based on current measurements and projections for silicon photonic ring-resonator modulators and receivers in the 2013-2014 time frame we employ 0.1 pJ/bit for the energy consumed at each of the E/O and O/E interfaces. In addition, the laser power consumption which is continuous is assumed to be 10mW per wavelength channel. Optical links are generally more power efficient at higher frequencies and typically operate today at 10 GHz or 40 GHz. However to simplify this calculation and to avoid additional computation of the serialization and de-serialization energies, we assume here that the optical signals run at the electronic tributary rate of 4 GHz. To achieve our designed bandwidth per super-core of 1 Tb/s, 250 wavelength channels are required. Although

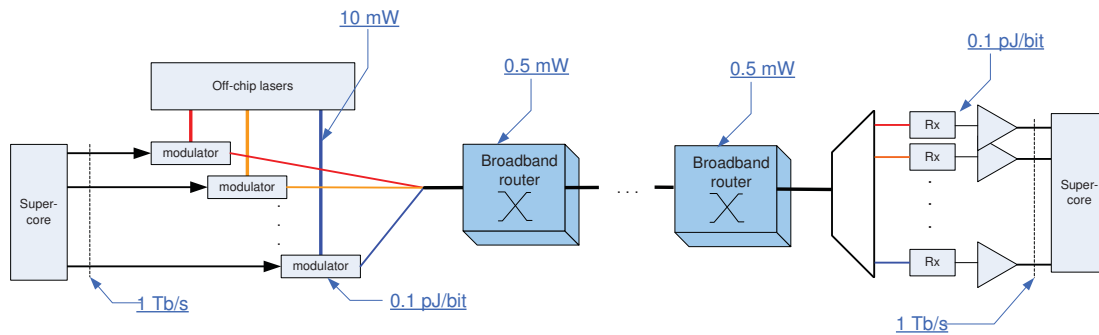


Figure 7.23: Super-core to super-core optical on-chip link.

Parameter	Value
Cores per supercore	21
Supercores per chip	36
broadband switch routers	36
Number of lasers	250
Bandwidth per supercore	1 Tb/s
Optical Tx + Rx Power (36 Tb/s)	7.2 W
On-chip Broadband routers (36)	18 mW
External lasers (250)	2.5 W
Total power for photonic interconnect	9.7 W

Table 7.14: Optical on-chip interconnect power consumption.

aggressive, this is not an unrealistic number, as it has been demonstrated within the context of commercial telecom fiber optic transmission. These components power parameters are summarized in Table 7.13. Each chip would require 250 external lasers each consuming 10mW for a total of 2.5W per chip.

Broadband silicon photonic router switches are used at each gateway to route optical messages within the on-chip network and to off-chip optical ports, as pictured in Figure 7.23. Based on today's silicon photonic switch technology, each router switch is assumed to consume approximately 0.5 mW. Importantly, this is independent of the number of wavelengths being routed simultaneously or the bit rate in each channel.

We now have all the components necessary to estimate the power consumed by the chip itself. Table 7.14 summarizes the calculations of the power consumed by on-chip photonic interconnection network.

There are several key observations from this simplified analysis. The total power of the on-chip photonic interconnect, estimated at 9.7 W is equivalent to the on-chip electronic interconnect in the strawman design of 8.4 W (Table 7.7). The photonic interconnect provides about a factor of 28 higher bandwidth (36 Tb/s versus 1.28 Tb/s) to the chip in comparison with the electronic on-chip interconnect. Secondly, the broadband router switches consume practically negligible power. As expected, the dominant contribution to the power dissipation comes from the E/O and O/E interfaces. Once these are performed however, the optical signal does not need to be regenerated

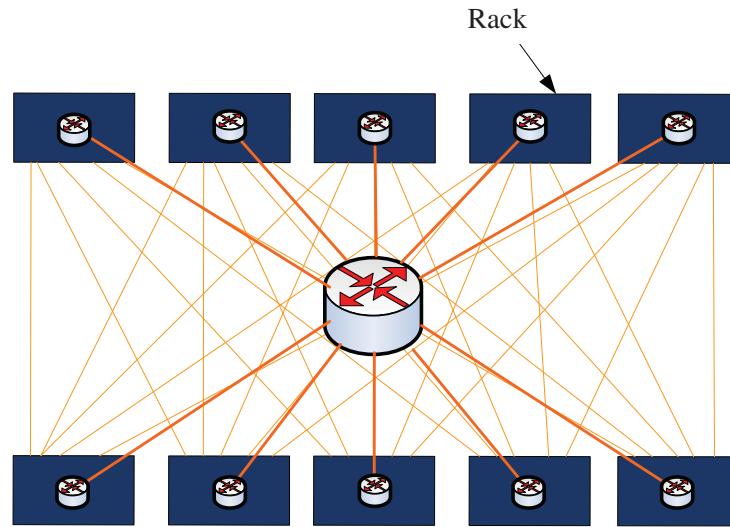


Figure 7.24: Optical system interconnect.

as it propagates off-chip and across the systems. This is the critical advantage that photonic interconnects can provide and will become apparent as we continue to compute the total Exascale system interconnect power dissipation.

From the strawman design, each processing node contains 16 chips of 1GB of DRAM each at 5.5 GWords/s. The total bandwidth of memory for each node is therefore 88 GWords/s, or 5.63 Tb/s. In this analysis we assume that the same optical Tx/Rx interface technologies are employed for the memory gateways.

Approximately 40 wavelength channels (or lasers) each operating at 4 Gb/s are needed to achieve this bandwidth across the 36 super-cores. This puts the power for the memory gateways per node: $40 * 10 \text{ mW} + 5.63 \text{ Tb/s} * 0.2 \text{ pJ/bit} = 1.526 \text{ W}$. It is clear that if we were to use a much higher bandwidth at the memory interface, such as 4.5 TB/s to match the bandwidth at the core, the memory interface power consumption would be similar to the on-chip interconnect or approximately 9.7 W.

The power to access DRAM was estimated at 10 pJ/bit for the strawman design. This includes the access and transmission of the data. It is unclear how much of this is contributed by each, so it is difficult to estimate the fraction of power contributed by the DRAM itself and the fraction that would fall under the optical interconnect budget.

7.5.3 Rack to Rack Optical Interconnect

At the system level, Figure 7.24, there are 583 racks each containing 384 processing nodes. We assume here that the racks are connected by a transparent optical network using a double layer hierarchical network topology with 2 optical MEMS 384x384-port routers per rack. These routers consume 100 W of power each, regardless of bit rate per port. We note here that this estimate is based on current optical MEMS cross-connects designed for telecom applications and is expected to be significantly less (by a factor of 10) when customized for computing systems. Racks are organized into approximately 60 groups of 10 with a central router, as shown in the figure below. The racks are also fully connected point-to-point to avoid the central router becoming a bottleneck

Parameter	Value
Bandwidth per chip	36 Tb/s
Bandwidth to/from memory per node	5.6 Tb/s
Number of MEMS routers	1226
Total Power for chip interconnect	9.7 W
Total Power for external network	122.6 KW
Total Power for node memory interface	1.53 W
Total power consumed by optics	2.6 MW

Table 7.15: Optical system interconnect power consumption.

for close-proximity transfers, which should be exploited in the mapping of the application to the system architecture. Note that because both the NoC and external optical network are integrated and transparent, data generated from one super-core on one chip in one rack traveling to any other super-core in the system will consume power per bit only at the gateway interfaces, set at 0.2 pJ/bit.

The only remaining power calculations are to add the power from the 1226 ($583 \times 2 + 60$) routers to the total power from the racks. Table 7.15 summarizes these power calculations.

7.5.4 Alternative Optically-connected Memory and Storage System

Exascale memory and storage systems must mitigate the disparity in bandwidth among levels of the hierarchy, achieve adequate storage performance, minimize energy expenditure, and provide adequate resilience. As discussed above, familiar DRAM main memory, disk, and tape hierarchy will probably not suffice. Memories in the form of commodity-based single-die silicon substrates will not deliver the requisite bandwidth and latency at acceptable power levels. Copper-based interconnect up and down the hierarchy may be inadequate to deliver the performance needed within the power budgeted. A truly serious approach to build an Exascale system within a decade may very well require alternative structures such as photonic communication between logic and 3D optically connected memory modules (OCM). Such a strawman memory system is discussed here.

As a baseline for a strawman data center scale memory system we assume a 32 petabyte main memory. Depending on the types of memory available, one may be able to afford more than that, in both cost and power terms, through the use of an OCM technology. data center projections shows storage file system to main memory ratios fall typically in the 100:1 range, and hence 4 exabytes of usable file storage is a reasonable estimate (adding 20 percent for ECC and other metadata). A substantial amount of storage needs to be much higher bandwidth and lower latency than current spinning disk technology, which is not improving for either of these performance measures.

Application checkpoints through defensive I/O to disk files play an essential role in ensuring resilience in today's Petascale systems. The growing disparity between disk bandwidth and integrated devices argues for more research into solid-state distributed storage solutions that are much faster and more energy efficient than mechanical systems. Several nonvolatile RAM (NVRAM) technologies look promising, as discussed previously. Some combination of DRAM and NVRAM in the same stack will significantly improve bandwidth and save energy. Depending on the cost and capability of the NVRAM, it may be able to supply most of the storage capabilities for an Exascale system, or at least act as a high capacity buffer to enable a better match between memory and

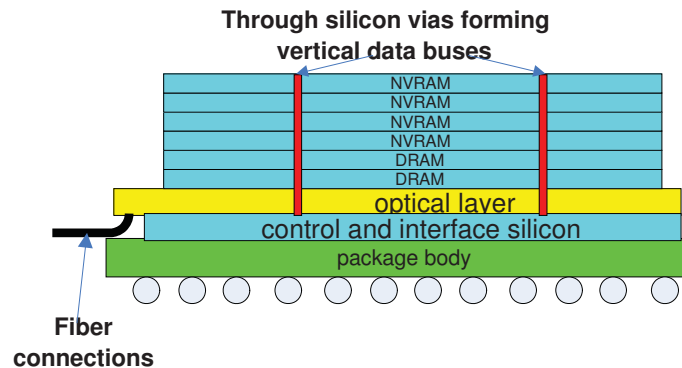


Figure 7.25: A possible optically connected memory stack.

storage.

The ITRS projects that each high performance DRAM die in a 2017 (20 nm) process will cost .03 microcents/bit with a capacity of 1-2 GB per die. A 32 PB commodity memory would therefore require 16.7-33 million die, each with 81-270 pins. The cost for just the memory die would be \$86.5M. The cost in power and DIMM packaging would increase this number significantly given the need for somewhere between 1.3 and 4.5 million wires. An OCM package such as that shown in the figure below that combines two layers of DRAM and four layers of high density NVRAM can reduce the number of components by at least 6, since the wires to each OCM module can be replaced by shorter in-module vias, and the external connection will only need 4 waveguides per OCM. Interconnect power alone for an Exascale system can be reduced from 15 MW (for all-electrical) to less than 1 MW for an optically interconnected system. A simple analysis for the DRAM portion of the OCM shows that at peak access rates, an 8 gigabyte OCM will consume 2.0 watts in 2017 and a 4 gigabyte OCM will consume 1.4 watts. This puts the total memory power requirement between 8.5 MW and 12 MW.

The OCM module, with a capacity of 4 or 8 gigabytes is constructed from a 3D stack of DRAM and NVRAM dies as shown in Figure 7.25. Each processor socket can connect optically to 64 of these modules. Using a photonic scheme enables each socket to be connected to more memory than would be possible by stacking the memory directly on top of a processor chip, and also avoids the problem of heating of memory by the processor when memory is stacked directly on top of a processor chip. More OCMs can be attached to the same memory controller channel for larger configurations. The channel consists of four waveguides, each of which carries 64 wavelengths modulated at 10 GHz for a total per channel memory bandwidth in excess of 160 gigabytes/sec per OCM, and 10 terabytes/sec per processor socket. Power and performance issues motivate an OCM organization where each access is a complete 128 byte cache line.

NVRAM capacity should be at least twice that of main memory, since a failure may occur during a checkpoint operation. More should be provided if affordable. It makes sense to place the NVRAM in the OCM stack: this allows fast in-stack DRAM to NVRAM copy, which saves energy since data will not have to be transported over a global interconnect.

There are several technologies currently competing for ultra-dense NVRAM, as discussed in Section 6.3.5. Simple crossbar circuits on top of functional CMOS have been demonstrated with a local density of 10 GB/cm² for a single layer. Further, the crossbar technology itself is stackable, so that effective densities as high as 100 GB/cm² on a single die are conceivable. There is a significant amount of activity in examining the switch material for nanometer-scalable resistive-RAM (RRAM)

junctions incorporating metal oxide layers (NiO and TiO₂), that in principle can be written in less than 10 ns for less than 1 pJ/b energy dissipation. The resiliency of these systems is being designed to handle 1-10% defects in the crosspoints.

7.6 Aggressive Operating Environments

By the end of the next decade, expected technology trends and possible architectural responses to exploiting them will impose extreme demands on software operating environments which will subsume the current responsibilities of run-time systems, operating systems, virtual machine layers, and system level schedulers and resource managers. As a consequence, they may look very different from today's operating environments. This section summarizes the extreme demands that will be placed on the operating environment software across the trans-Exaflops performance regime and discusses some elements of one possible approach to addressing them. Through this perspective, a set of general research questions is derived that may help.

7.6.1 Summary of Requirements

From this study, potential properties for Exascale computing systems has been explored and estimated. Here we highlight those that will have the largest impact on operating environments:

- **Parallelism** - concurrency of action will have to be elevated to unprecedented levels to reduce execution time, achieve target performance levels, and maintain adequate efficiency. It appears that sustained parallelism of a minimum of 100 million-way will be required to meet this requirement, regardless of technology. But time distances (measured in processor core cycles) to almost every resource whether local memory, remote nodes, or system services must also rely on additional parallelism to hide such latencies. While many factors contribute to determining exactly what the level of additional concurrency required for this purpose is, two orders of magnitude may prove a conservative estimate. The result is that future environments should be assumed to support parallelism of ten billion-way or more.
- **Latency** - As touched on previously, latencies measured as delay time in processor core cycles to intra-node and remote resources can be a source of extreme performance degradation. Multi-ten thousand-way latencies will demand locality management and latency hiding methodologies far more aggressive than today's systems. The apparent latency will be aggravated by contention for shared resources such as memory bank access and global system interconnect.
- **Overheads** - While conventionally a combination of static scheduling, global barrier synchronization, and large point to point data block transfers, future computations on Exascale systems may be very different. Application algorithms of future large scale problems will be multi-discipline, sparse, dynamic, and potentially irregular in data structure and operations on them. A major consequence is that the run-time resource management and application task synchronization overhead will become a significant part of the total execution.
- **Reliability** - As the number of devices grows and their size shrinks to nano-scale features, single point failures may have devastating effects on the operational viability of the system. Conventional practices of checkpoint/restart will be infeasible as the time to dump core to external storage will exceed the MTBF of the system. In the future, no system will be static in structure but will be continually modifying its form as hard faults dictate isolation of failed

components and transfer of data and tasks to functionally elements. An aggressive operating environment will out of necessity have to contend with this continuously changing physical platform while providing a global virtual interface to basic functional capabilities.

7.6.2 Phase Change in Operating Environments

To meet these unprecedented challenges, a change in execution model and operating environments is probable. While it is unclear at this time what form such a revolutionary environment may take on, it is possible to consider one such strategy as an exemplar for a class of possible alternatives. Such a strategy would mark a phase change in operating environments with respect to conventional practices as it directly addresses the challenges and requirements described above.

The levels of physical and abstract parallelism combined with the unprecedented potential efficiency losses due to overhead and latency will drive strategic changes. It is imperative that our objective function, our metric of success, be changed to reflect the realities of the costs and tradeoffs. Even today, we use as our principal measure of efficiency the ratio of sustained to peak floating point performance. In fact, the floating point ALU is among the least expensive resources in terms of die area, cost, or power consumption. Instead, the precious resources and therefore the overall efficiency are memory capacity, memory access bandwidth, and system bandwidth and latency. In addition, there should be high availability of flow control to handle over-subscription with respect to peak capability of these bottlenecks.

Dynamic scheduling of resources is required both for load balancing and reallocation in the presence of faults, as well as exigencies of system demand outside the user application domain. For this reason, the conventional practice of the user code controlling the dynamic scheduling is infeasible (especially when billions of threads of control may be present), and must be performed by a system sensitive set of policies and mechanisms while minimizing overhead impact on the application execution. In turn, this forces an important change in the relationship between operating and run-time systems, forcing a merger where there was once clear separation of roles and responsibilities. It also changes the relationship between the programmer and the resources. Where conventionally the programmer is largely responsible for explicit management of locality and resources, in the future the system operating environment will have to take on this responsibility, with declarative directives from the user indicating effects to be achieved rather than imperative statements of explicitly how to achieve them.

7.6.3 An Aggressive Strategy

An alternative operating environment to the incremental derivative described in Section 6.8 is suggested that may achieve superior scalability and robustness for dynamically adaptive systems while providing a relatively simple implementation strategy. The approach proposed employs lightweight kernel elements in each local collection of physical threads. Unlike typical lightweight kernel methods, the virtualization is not limited to each "node" but across the entire system through an intermediate set of protocols between kernel element instances. For simplicity, the brief description here is considered for three primary functionalities, although in a realistic implementation other support mechanisms would be provided as well. The three basic system level functions are: distributed memory, threads, and communication fabric.

The expected physical memory will likely comprise multiple distributed sets of local memory banks with a minimal manager for each fixed sized bank. The local memory kernel serves this resource in an object oriented manner through a defined protocol of functions. However, this protocol extends beyond the local memory bank to include interaction and possible negotiation

with other like local memory kernels serving separate memory banks. This synergistic operation permits collective functionality across local memory resources to satisfy relatively global memory demands. Such functionality includes dynamic data distribution, global address translation, fault management, and copy semantics.

The other two global virtual functions, distributed threads and communication fabric, are achieved through a similar global combination of local lightweight kernel elements. Within the operational framework under consideration, tasks in the form of lightweight threads are scheduled to be executed where their primary operand data are located and when there are available execution resources. Thus, an aggressive operating environment will support a global distributed thread execution environment where the work may move to the data when appropriate rather than the data always moving to the work, a potentially costly operation.

The communication fabric is also supported by the synergy of like lightweight kernel elements managing the flow of information, routing, and address translation. An important property of all three of these distributed functionalities is that they adapt to changing structures due to diagnosed faults and other drivers of reconfiguration. As a physical module is deemed faulty, its local kernel elements are turned off and the neighboring elements book-keep the fact that they no longer represent available resources, providing a degree of system adaptive reconfigurability.

The operating environment supports a message-driven work queue execution model in addition to the more conventional process oriented message-passing model of computation. This advanced strategy for controlling distributed processing exposes a high degree of parallelism through lightweight synchronization mechanisms and is intrinsically latency hiding. This is because the work queue model, assuming sufficient parallelism, does not wait on remote actions but processes a stream of incoming task requests on local data. This method also reduces overhead by localizing synchronization within the flow control or within the data itself, thus eliminating such unscalable techniques like global barrier synchronization. The strategy outlined here is highly scalable if the operation of the local lightweight kernel elements are primarily dependent in the state of their neighboring domains of local resources. The design of these kernel elements is simple. Complexity of operation is derived, not through complexity of design, but rather through the emergent global behavior of the synergistically interacting collections of simple local kernel elements. This greatly bounds difficulty of software design, debugging, and scalability.

7.6.4 Open Questions

An aggressive strategy to billion-way parallelism (or more) processing systems' operating environments presents many challenges and questions to be resolved before realization of the promise implied. The sketch of an aggressive operating environment above exposes a number of issues that must be resolved prior to realizing the promise of such future systems. Here we briefly discuss some of these key research questions.

- **Model of Computation** - The key issue driving all research for future Exascale computing system architectures is the principal model of computation that will provide the governing principles of parallel computation and the interrelationships among the physical and abstract computing elements. If the model is an immediate extension of current conventional practices based on a mix of communicating sequential processes globally and local shared memory operation, then locality management becomes paramount to avoid performance degradation due to latency and overhead. If, as has happened many times in the past, the field transitions to a new model better capable of exploiting the emergent technologies and enabling new classes of dynamic applications, then this enabling paradigm or perhaps a multiplicity of

such paradigms needs to be devised against a set of specified requirements. It should be noted that although a significant shift in methodology is suggested, this does not assume a disruptive discontinuity with respect to legacy application codes and techniques. Rather, any such innovation should subsume as a semantic subset the typical practices of the past.

- **Synergy Algorithm** - In most simple terms, the choice space for managing large scale resources is either a centralized control system or a fully distributed control system. Most systems today are either the former or a cluster of such systems; the extreme being Grid based widely distributed systems. There has been little exploration of the latter, fully distributed control systems. Yet, it is possible that the degree of scaling expected by the nano-scale era will make any centralized methodology infeasible for both efficiency and reliability reasons. A class of algorithms must be derived that enables and supports the synergistic interaction of simple local rule sets to achieve the desired emergent behavior of each of the critical global functionalities of the system yielding a global virtual machine that is dynamically adaptive to resource demands in the presence of faults. This resilient strategy is scalable to the regimes of parallelism demanded and should be the subject of future research. In this context, we will see a merger of operating system and run-time system roles and responsibilities which today are largely mutually isolated.
- **Global Name Space Management** - Whether merely at the user application level or as an intrinsic of the hardware architecture, every program exhibits a global or hierarchical name space; the set of referents to which the algorithm is applied. Conventional practices assume a physically fragmented name space on most scalable systems (DSM is an exception) at least to the level of the node. Load balancing is handled if at all by user transformation of effective addresses from one node to another; a low level and costly effort with many restrictions and highly error prone. Research that finds a superior mode of operation somewhere between full DSM and PGAS is required to guide future system design in support of user name spaces in dynamic adaptive resource management systems, in part due to the needs of resilience.
- **Fault Management Methodology** - New methods are required to provide effective functionality in systems of the scale being considered. It is recognized that conventional practices of checkpoint/restart will not scale to Exascale system structures, in part because the MTBF with single point failure modes will cross the threshold for which it will be shorter than the time to dump core. Research will be required to address this challenge by building in micro-checkpointing semantics directly in to the execution model and to provide low overhead and robust mechanisms to support it. Key is the management of commits of global side-effects to ensure that any such are performed only after verification of correctness. A second challenge is the dynamic remapping of virtual to physical addresses and its efficient implementation in system interconnect routing. This may require a second level of hardware address translation to complement conventional TLBs (translation lookaside buffers).
- **Programming Models** - Programming languages have served poorly in recent years as the programmer has been responsible for hands-on management of the low level resources as they are applied to the user application. Research is required to provide an API that serves the new class of operating and run-time system models developed in the context of dynamic adaptive resource management and application execution. While it is often thought that the user needs access to the lowest level to achieve adequate performance, an alternative strategy will have to be devised that supports hardware/software co-design so that the architecture, new run-time, and programming interface are devised to be mutually complementing. Multiple levels

of programming may be a resulting set of solutions depending on computational domain with source to source translation proving the norm to lower level APIs. Compilation strategies will rely more heavily on run-time resource management and thus become more simple. However, they will also have to contend with heterogeneous hardware structures and therefore require a more advanced protocol between compiler and run-time elements.

7.7 Programming Model

While that programming model for the strawman proposed in this section cannot be specified completely at this time, many of its properties may be considered with some confidence. Among these are:

- Expose and exploit diversity of parallelism in form and granularity;
- Lightweight (low overhead) local synchronization;
- Intrinsic latency hiding and locality management;
- Local/incremental fault management and graceful degradation;
- Global name space and virtual to physical address translation;
- Dynamic resource management and load balancing;
- Energy minimization per operation;

Other properties may be assigned as well in response to in depth consideration of the needs of device technology and system scale envisioned for the next decade.

In each of these areas, the development of new technologies is required, but in many cases quite feasible. For example, many of the features of the HPCS programming models are designed to expose parallelism in many forms. These features, and especially the tools behind them, can be enhanced for the even greater parallelism required for Exascale. Energy minimization models already exist in the embedded computing market and it is quite likely that their features could be applied here as well. The major challenge will be to produce a model which combines all of the features in a coherent and productive whole.

7.8 Exascale Applications

In 2007, Scientists and engineers from around the country have attended three town hall meetings hosted by DOE's **Simulation and Modeling at the Exascale for Energy, Ecological Sustainability, and Global Security** (E3) initiative. At these meetings, participants discussed the future research possibilities offered by Exascale supercomputers capable of a million trillion calculations per second and more. A primary finding was that computer scientists will need to push the boundaries of computer architecture, software algorithms, and data management to make way for these revolutionary new systems.

Typical exercises at this series of workshops included extrapolations of science problems of today to Exascale. Many of these exercises involved conceptually increasing the size and scope of the input data, adding new physics and chemistry to the calculations, increasing resolution, and coupling disparate system models.

The following subsections describes how several applications could plausibly be mapped to the kinds of systems extrapolated in Section 7.3. In all cases, the match indicates a reasonable to good match to the capabilities of the machine, and thus a reasonable claim of achieving “Exascale performance.”

7.8.1 WRF

Consider a futuristic calculation that might be carried out using WRF. The largest run of WRF ever carried out to date is described in [105]. The performance characteristics and a related performance model were presented in Section 5.7.2. The record calculation is 2 billion cells, 5km square resolution, 101 vertical levels on a half-hemisphere of the earth. Using rounded-off numbers for ease of projecting to Exascale, this calculation achieved about 10 Tflops on 10,000 5 GFlops nodes (about 20% of theoretical peak) on a system with 1 GB of memory per node and sustained memory bandwidth of about 1 byte per flop.

The strawman architecture of Section 7.3 would confer about a 10x increase in total memory capacity over the machine of today (this is the strawman’s least dimension of improvement). One could still increase the number of cells under simulation with WRF to about 20 billion, going down to about 1km square resolution on such a global calculation, thereby to capture such effects as cloud dynamics in the atmosphere and interaction with topography on the ground – a calculation of great scientific interest.

The strawman architecture would represent about an order-of-magnitude increase in the number of nodes over the number used in the record run. However given WRF’s inherent parallelism and the relative robustness of the strawman network, this should pose little challenge to WRF (indeed in unpublished work WRF has been run at 100K nodes on BG/L - the same order-of-magnitude in number of nodes as the strawman.)

Although the strawman represents about a 1000x increase in peak flops per node, it delivers only about a 100x increase in memory bandwidth per node. WRF is memory bandwidth limited (see Section 5.7.2). Efficiency (percentage of peak) could then fall an order-of-magnitude (to 2% from 20%).

Reading the performance prediction for WRF to improvements in flop issue rate and memory bandwidth (more exactly $1 / \text{memory latency}$) off of Figure 5.11 one should then be able to run the 10x larger problem 100x faster than on today’s machines (if today’s machine had the memory capacity). This is a perfectly plausible definition of Exascale as a 10x larger problem 100x faster (a 1000-fold increase of today’s capability).

This above projection is the optimistic one based on the notion communications overheads will not grow as $\log(n)$ of node count (not implausibly optimistic given WRF’s locality, and likely increased computation to communication ratio for the more highly resolved problem). A more pessimistic projection could be read off of Figure 5.14 (25x faster on the 10x bigger problem), but that still represents a 250x capability boost for WRF.

7.8.2 AVUS

Consider a futuristic calculation that might be carried out using AVUS[26]. The performance characteristics and a related performance model were presented in Section 5.7.2.

A future calculation might model the entire structure of a full aircraft interacting with atmosphere and sound waves hypersonically (next generation fighter) under maneuvers. This problem maps plausibly to an order-of-magnitude increase in memory footprint (current calculations focus

typically on a part of the aircraft i.e. wing, tail, or fuselage) so a 10X memory capacity boost allows full aircraft model to be held in memory with useful resolution.

Like WRF, AVUS is not highly communication bound, and is quite scalable by weak scaling (making the problem larger). It is however even more memory-bandwidth-bound than WRF. Reading the performance prediction for AVUS to improvements in flop issue rate and memory bandwidth (more exactly the reciprocal of memory latency) off of Figure 5.12 one should then be able to run the 10x larger problem 70x faster than on today's machines (if today's machine had the memory capacity).

This above projection is the optimistic one based on the notion communications overheads will not grow as $\log(n)$ of node count (not implausibly optimistic given AVUS's locality, and likely increased computation to communication ratio for the more highly resolved problem). A more pessimistic projection could be read off of Figure 5.14 (50x faster on the 10x bigger problem - AVUS is even less communications dependent than WRF) still a 500x capability boost for AVUS.

7.8.3 HPL

Recall from Section 5.7.2 that even HPL has some memory references and some slight dependency on memory bandwidth. Reading off projected performances from Figure 5.13 it is predicted that a machine such as the strawman could run an HPL problem 10x larger than one that can be solved today 125x faster than today, a greater than 1000x boost in capability.

7.9 Strawman Assessments

This chapter has provided a series of insights that seem directly relevant to achieving Exascale computing technology, including:

1. Silicon technology has reached the point where power dissipation represents the major design constraint on advanced chips, due to a flattening of both chip power maximums and of V_{dd} .
2. In terms of chip microarchitecture, the above constraints have driven the silicon design community towards explicit parallelism in the form of multi-core processor chips, with flat or even declining clock rates.
3. For Exascale systems, power is perhaps the major concern, across the board. Real progress will be made when technical explorations focus not on power, but on "energy per operation" in regimes where there is still enough upside performance (clock rate) to moderate the explosive growth in parallelism. For silicon, this appears today to lie in low voltage, but not sub-threshold, logic running around 1-2 GHz in clock rate.
4. From an overall systems perspective, the real energy (and thus power) challenges lie not so much in efficient FPUs as in low energy data transport (intra-chip, inter-chip, board to board, and rack to rack), and in the accessing of data from dense memory arrays.
5. DRAM memory density growth is slowing, because of both a flattening in the basic bit cell architecture and because of growing concerns about the increasingly fine features needed within the memory arrays. In fact, while in the past, DRAM has led the industry in improving feature sizes, it is now flash that will drive DRAM.

6. The voltage used with the DRAM memory structures has long since flattened at a much higher level than that for logic of any kind, meaning that the energy per bit accessed inside commercial products will see little decline in the future.
7. Modern commodity DRAM chip architectures have major energy inefficiencies that are built in because of the protocols that have developed to communicate with them. With current memory system architectures, the transmission of addresses and commands is replicated to multiple chips, with each chip then internally accessing and temporarily storing much more data than is passed to the outside world. While there appears to be nothing technologically that stands in the way of rearchitecting DRAMs into a much more energy efficient form, the cost-driven nature of the commodity DRAM business will preclude that from happening on its own accord.
8. Flash memories hold significant density advantages over DRAM, but their currently relatively high write power and relatively limited rewrite lifetimes preclude their serious use in Exascale systems. However, it does appear that both problems may be solvable by relaxing retention times. This, however, requires rearchitecting both the devices and the chip architectures, something that is difficult to do in the cost-driven commercial market.
9. A variety of novel non-silicon devices and chip architectures have been proposed, with perhaps the greatest potential coming from those that can implement dense non-volatile memory structures, particularly ones that can be built in multiple layers, especially above conventional logic. As with the optical devices, however, there is still significant development work needed to bring them to commercialization, and significant architectural work to determine whether, and how best, to marry them with conventional technology.
10. A variety of novel on-chip optical devices have been prototyped, but before definitive statements can be made about their real potential, complete end-to-end energy per bit cost estimates must be made and compared to advanced all electrical protocols in a full system context. This includes the electrical costs of serializing parallel data from a conventional core to a high speed serial stream (and then back again at the other end), the fixed overhead costs of the light sources, the costs of switching the photonic routers (especially those distant from the source and to which routing information must be sent), and in providing appropriate temperature control for the optical devices, especially as large numbers of wavelengths are employed. In addition, these devices are just now being prototyped at low levels of integration, and there is still significant work that needs to be done to complete their characterization and optimize their designs and architectures for commercialization, especially by 2015.
11. Conventional spinning magnetic disk drives continue to advance in density, although latency for random accesses has been flat for years, and data rates, while growing, are still no match for solid state memories of any kind. However, at least through 2015 they seem to continue to hold an edge in overall physical densities over alternative emerging mass storage technologies.
12. A variety of alternative chip packaging and cooling technologies are emerging that may prove useful in moving memory and logic closer together, particularly in ways that lower the energy per bit transported, and thus result in significantly lower system power. Leveraging such technologies, however, requires rearchitecting the underlying chips.
13. Both fault mechanisms and fault rates will degrade as we go forward. Silicon below 45 nm will begin to exhibit more instabilities and wearout mechanisms that will be exacerbated

by lower voltages with less margin. Many of these effects will be temperature and/or time-dependent, such as the variable retention bit problem being observed today in DRAMs. All these, especially when coupled with the massive numbers of components in the data center scale systems, will introduce more complex failure patterns and higher FIT rates into system designs.

14. From the three strawmen system architectures explored, the heavy weight strawman based on leading edge commercial microprocessors, the light weight strawman based on lower power and simpler multi-core scalable compute chips, and the aggressive design based on low voltage logic and minimal overhead, the following observations are made:

- Designs based from the beginning on massive replication of small chip sets with tailored and balanced design characteristics are by far more energy (and thus power) efficient. This was obvious from the light weight and the aggressive strawmen projections.
- Reducing energy overhead costs in CPU microarchitecture is a necessity, and results in very simple cores that are then massively replicatable on a processor die.
- Reducing the on and off-chip data transport costs is crucial, with low swing signalling a necessity.
- A combination of memory chip architectures, off-chip contacts, and chip-to-chip data transport energy costs will tend to keep the number of DRAM die that can be supported by a single processor die in a dense configuration to a relative handful, meaning that the intrinsic bytes to flops ratio of future Exascale systems is liable to be significantly lower than that seen on classical computers.
- The aggressive strawman design is not possible without a rearchitecture of the DRAM chips to solve the above problems, and a packaging scheme that allows lower energy chip-to-chip transport to occur.
- Integrating significant routing capabilities into each processing die seems to pay significant dividends in reducing overall system power, especially if high bandwidth pGAS systems are desired.
- Regardless of architecture, massive concurrency that is largely visible to, and must be expressed by, the program seems inevitable. When overheads for latency management are included, the total number of threads that an application may have to express for execution on data center scale problems will reach into the billions.
- This explosion in concurrency will also exhibit itself at both the departmental and embedded scale. The numbers present at the departmental scale will rival those expected in the near-term Petascale systems, meaning that such systems will be unusable unless the heroic programming efforts needed for today's supercomputers can be greatly simplified. While not as severe at the embedded scale, there still will be the emergence of the need to express embedded applications in a several hundred-way parallel fashion, something that is not common today.
- There are at least a few applications such as WRF, AVUS, and HPL, that appear scalable to an "exa" level at reasonable efficiencies, even with the relatively low bytes to flops ratios that are liable to be present.

Chapter 8

Exascale Challenges and Key Research Areas

The goals of this study were two-fold: determine what are the major technological barriers to enabling Exascale systems by 2015, and suggest key research directions that should help accelerate the reduction or elimination of these barriers.

As reference, Figure 8.1 (a variant of Figure 7.11) places one possible specific goal of an exaflops in a data center class system in the context of the current Top 500 projections from Section 4.5, and the evolutionary strawmen projection of Section 7.2. Assuming that Linpack performance will continue to be of at least passing significance to real Exascale applications, and that technology advances in fact proceed as they did in the last decade (both of which have been shown here to be of dubious validity), then while an Exaflop per second system is possible (at around 67MW), one that is under 20MW is not. Projections from today's supercomputers ("heavy weight" nodes from Section 7.2.1 and "lightweight" nodes from Section 7.2.2) are off by up to three orders of magnitude. Even the very aggressive strawman design of Section 7.3, with processor, interface techniques, and DRAM chip designs we have never tried commercially at scale, is off by a factor of greater than 3 when an upper limit of 20MW is applied, and this is *without* some major considerations discussed in Section 7.3.9 such as a very low memory to flops ratio. This gap will get even worse when we consider more stressing applications.

While this result is disappointing at the data center scale, the aggressive strawman design by itself does indicate at least a 3X better potential than the best of the extrapolations from current architectures – a significant advance in its own right. Further, a 3X improvement in terms of power at the departmental scale is of potentially tremendous commercial value, since it would mean that today's data center class Petascale system will fit (aggressively) in 2015 into a very small number of racks. This in turn has the breakthrough potential for growing the Petascale user community many-fold.

Likewise at the embedded level, the techniques that make the aggressive strawman possible seem to offer the potential for an order of magnitude increase in power efficiency over today. This alone will enable a huge increase in embedded computational potential.

The conclusion from this is that regardless of the outcome of the data center, the results of this study indicate that if the challenges described here can be met, then there is a very significant payoff potential across the board. In particular, the study group's conclusion was that there are four such very specific major consensus **challenges** for which there is no obvious technological bridge with development as usual, and/or that will need particular attention to ensure that they do not rise to the level of a showstopper. These challenges focus on energy, memory, concurrency,

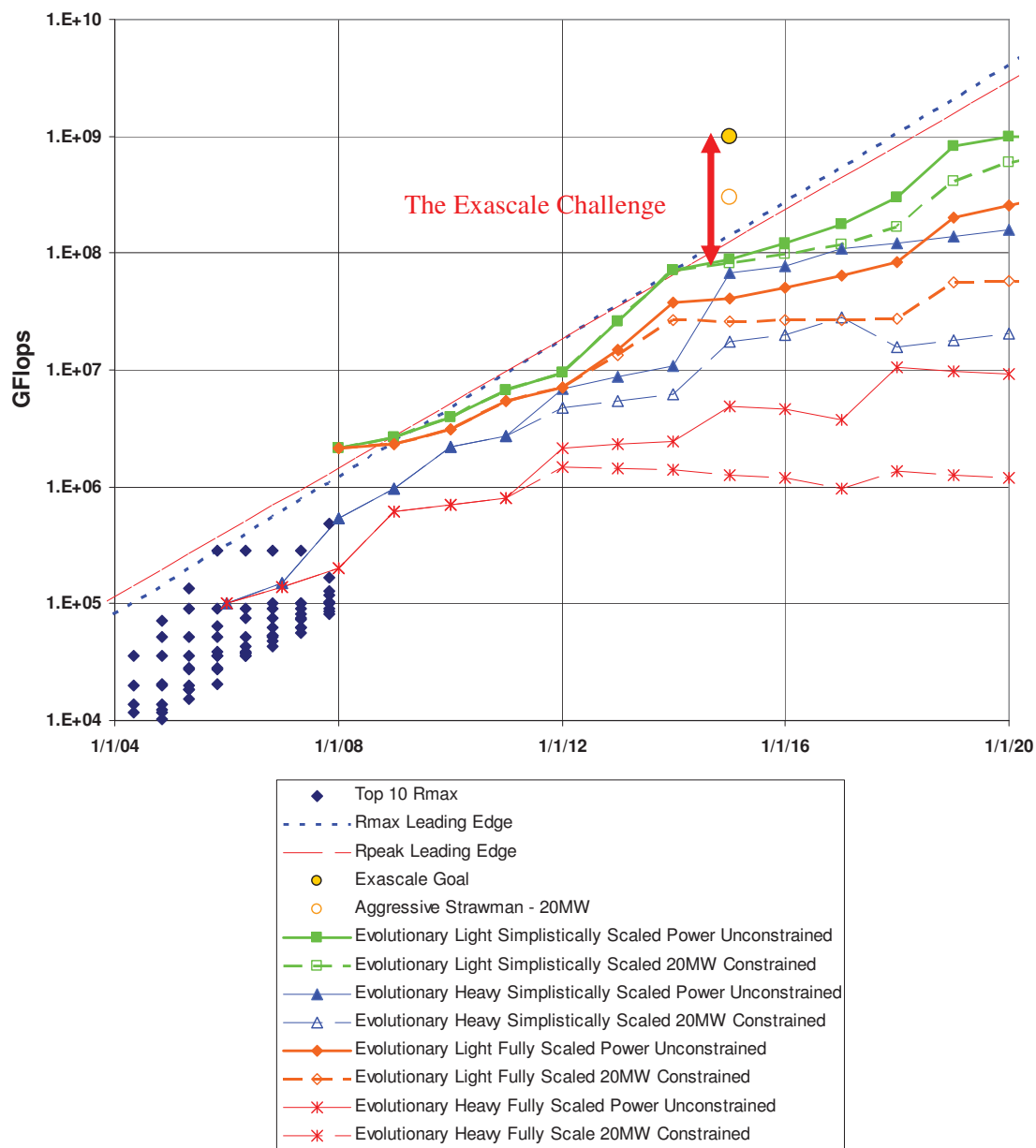


Figure 8.1: Exascale goals - Linpack.

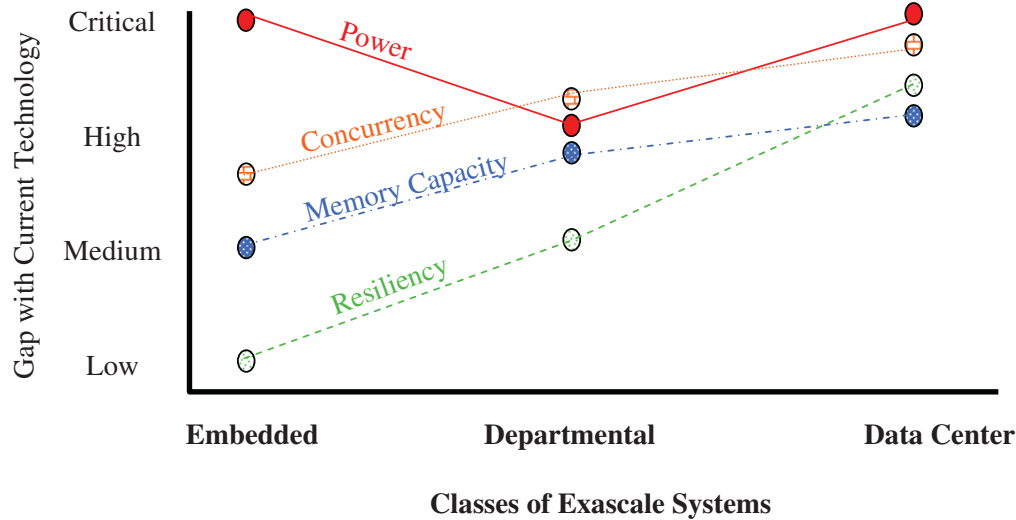


Figure 8.2: Critically of each challenge to each Exascale system class.

and overall system resiliency. Each of these challenges are discussed in greater detail in Section 8.1. However, Figure 8.2 diagrams notionally how important each of these challenges is to each class of Exascale system. As can be seen two of them, power and concurrency, are or real concern across the board, while the other two become of increasing importance as the size of the system increases.

The study group also developed a list of research thrust areas (Section 8.2) where significant progress in the next few years could go a long way towards reducing the barriers caused by the challenges, and thus enabling Exascale systems. Table 8.1 summarizes the relationship between these barriers and the proposed research directions. Its columns (challenges) are discussed in more detail in Section 8.1, and the rows (thrusts) in Section 8.2.

8.1 Major Challenges

As was demonstrated in Chapter 7, there are multiple areas where the natural progression of technology appears inadequate to enable the implementation of systems that come anywhere achieving Exascale attributes in the desired timeframe. Each of these thus represent a major **Challenge**, and is discussed individually below.

8.1.1 The Energy and Power Challenge

The single most difficult and pervasive challenge perceived by the study group dealt with **energy**, namely finding technologies that allow complete systems to be built that consume low enough total **energy per operation** so that when operated at the desired computational rates, exhibit an overall **power dissipation** (energy per operation times operations per second) that is low enough to satisfy the identified system parameters. This challenge is across the board in terms of energy per computation, energy per data transport, energy per memory access, or energy per secondary storage unit. While there has been a recognition of this challenge before, the focus has been predominately on the energy of computation; the real impact of this study is that the problem is much broader

Exascale Research Thrust	Challenges							
	Power & Energy		Memory & Storage		Concurrency & Locality		Resiliency	
	Crit	Gap	Crit	Gap	Crit	Gap	Crit	Gap
Technology & Architectures	High	High	High	Med			High	Med
Architectures & Programming Models	Med	Med			High	High	High	Med
Algorithms & Applications Development	Low	Med	Med	Med	High	High	Low	High
Resilient Systems	Med	Med	Med	Med			High	High
Crit. = criticality of thrust area to the Challenge for widespread solutions. Gap = the gap between the maturity of existing research and the needed solution. A “Med.” in the Hardware row reflects existence of lab prototype devices. Blanks for any entry imply that the interaction is indirect.								

Table 8.1: The relationship between research thrusts and challenges.

than just “low-power logic” - it truly is in the entire system. In fact, in many cases it has become clear to the panel that the non-computational aspects of the energy problem, especially the energy in data transport, will dwarf the traditional computational component in future Exascale systems.

While the resulting power dissipation is a challenge for all three classes of Exascale systems, it is particularly so for the largest data center class. The design target of 20MW for the electronics alone was chosen to have some flexibility above that of today’s largest systems, but still not be so high as to preclude it from deployment in anything other than specialized strategic national defense applications. Figure 8.3 presents some historical data along with a “trend line” and the Exascale goal assuming a Linpack reference. The reference metric in this case is “Gflops per Watt,” where the power is taken over the entire system, not just the floating point units. As can be seen, even if the positive trends of the last two decades were capable of being maintained, in 2015 power would still be off by between a factor of 10 and 100.

As discussed in Section 7.2.1, for at least a “heavy node” system architecture, although the most optimistic of possible projections barely makes the current trend line, more realistic estimates seem to be barely more than flat from today. The “light node” system architecture is better, but still is a factor of 10 off. Even the very aggressive strawman design of Section 7.3, with processor and interface techniques and DRAM chip designs we have never tried commercially at scale, is off by a factor of greater than 3, and this is *without* some major considerations discussed in Section 7.3.9 such as a very low memory to flops ratio.

As discussed in the roadmaps of Chapter 6, a variety of factors are responsible for this effect, especially the flattening of V_{dd} and the concurrent flattening of clock rates, that in turn force performance growth by physical parallelism alone and increasing power-consuming area. Further, as the point studies and strawmen of Chapter 7 indicated, even aggressive circuit and architecture approaches to lowering energy do not close the gap in a timely fashion.

The following subsections discuss individual components of this power and energy challenge in detail.

8.1.1.1 Functional Power

First, if a *peak exaflop* per second was the metric (i.e. a Linpack-based extrapolation of today’s Top 500 rankings), then silicon based floating point units (FPUs), by themselves, exceed 20 MW by

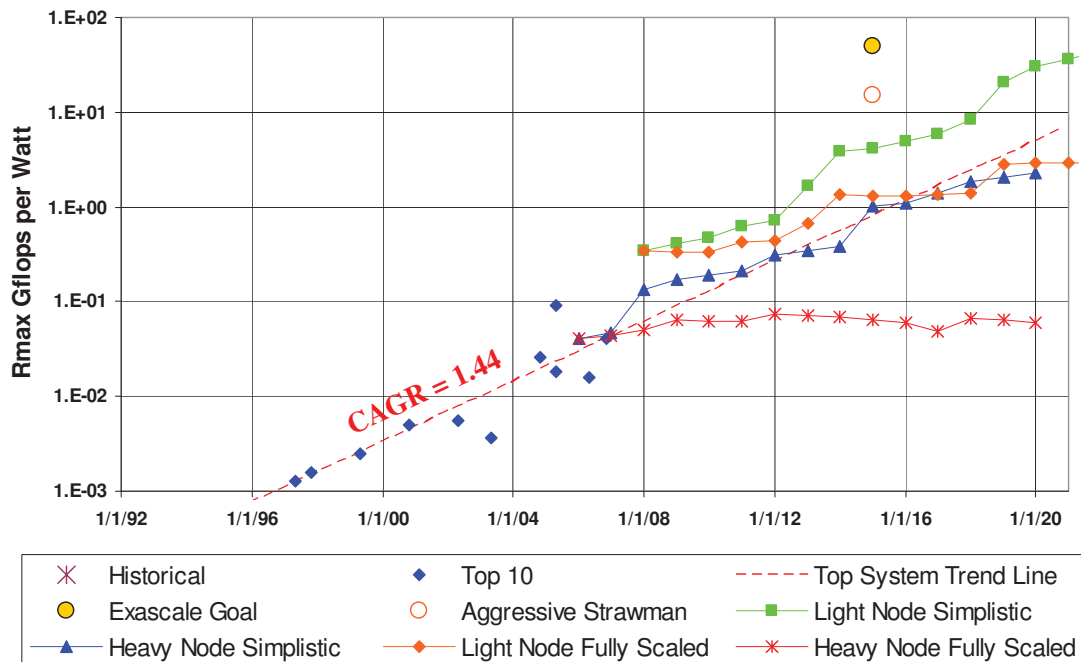


Figure 8.3: The power challenge for an Exaflops Linpack.

themselves using today's high performance logic (Figure 7.1). We are still around 10 MW even if aggressive circuit designs and lowered clock rates were used that reduced V_{dd} to 2X the threshold, and then somehow are used at 100% efficiency - regardless of the resultant explosive growth in required parallelism to literally hundreds of millions of simultaneous operations.

Again, as discussed in prior chapters, both the probability of 100% efficient use of numbers of FPUs far in excess of anything we've managed to date, and the applicability of floating point operations alone as metrics for successfully deployable Exascale applications are indicators that this problem is in reality even more severe than depicted.

8.1.1.2 DRAM Main Memory Power

Next, if DRAM is used for main memory, then its power is a function of:

- Total memory capacity: this includes the continuous energy needed for refresh.
- Total number of independent accesses that must be maintained per second: each memory access that escapes any cache hierarchy must activate some memory bank on some memory chip to retrieve the necessary data.
- The number of bits read per access versus the number of bits actually transferred.
- Data bandwidth needed to move the accessed data from the on-chip memory banks to the off-chip contacts of the DRAM (regardless of how such DRAM are packaged).

As discussed in Section 7.1.4, such numbers may be reasonable for very small memory capacities and non-memory-intensive applications, but rapidly run out of control when either memory

capacities exceed around 5PB or applications such as GUPS are the driving factors.

8.1.1.3 Interconnect Power

Interconnect power comes in several forms: on-chip, chip to nearby chip, board to board, and rack to rack. Even with a variety of circuit techniques (such as low-swing on-chip interconnect), emerging chip-chip techniques (such as through via), and optical signalling techniques, the energy to move one bit through any one level was on the order of 1-3 pJ (Section 6.5). For any one level of the transport path that requires on the order of an exabyte per second, the power thus balloons to 10-30MW, and multiplies as multiple types of interfaces must be crossed by the same data.

8.1.1.4 Secondary Storage Power

Today the only media for scratch and persistent storage is disk, and projections for the highest density per watt in the 2014 timeframe are for up to 5MW for each exabyte (Section 6.4.1.2). Further, the actual amounts of such storage needed is a function of the main memory capacity, which if system implementations move into the larger regimes to support emerging and stressing applications, may result in upwards of 100EB of such storage, as discussed in Section 7.1.6.

8.1.2 The Memory and Storage Challenge

The second major challenge is also pervasive, and concerns the lack of currently available technology to retain at high enough capacities, and access information at high enough rates, to support the desired application suites at the desired computational rate, and still fit within an acceptable power envelope. This information storage challenge lies in both **main memory** and in **secondary storage**. By main memory we mean the memory characterized by the ability of a processor to randomly access any part of it in relatively small increments with individual instructions. By secondary storage we mean the memory where data is typically accessed in “blocks” with access handled by subroutines, not individual instructions. This includes both **scratch storage** for checkpoints and intermediate data files, **file storage** for more persistent data sets, and to some extent **archival storage** for the long term preservation of data sets.

While this challenge is felt by all Exascale classes, it is particularly severe at the data center scale, where affordable capacities are probably at least an order of magnitude less than what is needed for the projected application suite.

8.1.2.1 Main Memory

DRAM density today is driven by both the architecture of an individual bit cell (and how the capacitor that stores the information is merged into the access logic) and the basic metal-to-metal feature size of the underlying level of lithography (how close can individual row and column access wires be placed). Today, and for the foreseeable future, the architecture of a basic cell will be stuck at $6F^2$, where F is the technology feature size (1/2 pitch of M1). Second, this feature size is driven today not by DRAM but by flash memory technology, and there are serious concerns as to how this can be achieved below 45 nm. Together, this makes the ITRS projections[13] of 1GB per commodity DRAM chip in the desired timeframe rather aggressive. However, even at 1GB per chip, each PB of main memory translates into 1 million chips, and with realistic capacity needs for data center class systems in the 10s to 100s of PB, the number of such chips grows excessively, resulting in multiple power and resiliency issues, not to mention cost.

Additionally, there is a significant challenge in bandwidth, that is how to get enough data off of each memory die to match the desired rates. While as shown in Section 7.1.4.2, it is possible with projected silicon technology to provide enough such bandwidth, the resulting chips look nothing like the commercial high-volume parts of today, or the next decade. Thus even if DRAM capacities were sufficient, there is a significant challenge in how such chips should be organized and interfaced with other system components, and in how such chips could be brought to market in a way that is economically competitive with today's commodity DRAM.

The challenge thus is to find a way to increasing memory densities and bandwidths by orders of magnitude over that which is projected for 2014, without running into other problems. The study considered flash memory in various forms as an existing technology that might be employed somehow, since it has both significant density and cost advantages over DRAM. However, its slow access times, and limited rewrite lifetimes made it unsuitable for at least the fast random access part of the main memory role. The study did, however, encounter several other emerging non-silicon memory technologies, as described in Section 6.3.5, that have the potential for such density gains, but not on a commercialization path that today will result in useable devices in the appropriate time frame. Further, it is unclear how best to architect memory systems out of such devices, as replacements for DRAM, or perhaps as a new level of memory within the overall information storage hierarchy.

8.1.2.2 Secondary Storage

As discussed previously (Section 6.4.1), current scratch and file systems have been implemented with the same disk technology that has Moore's law-like growth in density, and some increase in bandwidth, but has been essentially stagnant in seek time for decades. Also, as described in Section 5.6.3, growth in storage requirements above the main memory level has been continuous, with scratch needs growing at a rate of 1.7X to 1.9X per year, overall projections for scratch in the 20-40X main memory, and that for file systems in the 100X range.

For data center class systems, projected disk storage density growth using these factors is acceptable as long as the implemented main memory is in the low petabyte range. However, there are significant Exascale applications with needs way beyond a few petabytes that would in turn make achieving sufficient secondary storage a real difficulty, particularly in complexity and in power.

While flash memory may have a role to play here, flash as currently designed does not have a sufficient level of rewrites to make it acceptable as is. The alternative memory technologies mentioned as possibilities for main memory are also possibilities, but again there are currently significant challenges to making them viable enough, with the right systems architectures, to take on such replacement roles.

A second consideration deals again with bandwidth. For scratch storage the need to do checkpointing requires copying the bulk of main memory to disk, usually with the application suspended in the process. Today, for memory-rich systems this process often takes up to 50% of the execution time, and with the stagnation of disk bandwidth, this fraction could grow even higher, leaving no time for computation to advance before another checkpoint is required. Thus, while extrapolation from the strawman indicates that with the sheer number of disks needed to provide such backup for the 3-4PB main memory range may provide sufficient bandwidth, this may not hold true if main memory needs on a per operation basis grow to ratios commensurate with today's systems.

Another major concern with storage is with the growing application-level need to perform small unaligned I/O. Because of the flat seek times foreseen for future disks, achieving the peak bandwidths assumed above can only be done by performing all transfers in large megabyte or bigger sized blocks, and aligned with the basic disk block size. Unfortunately, many of today's

more critical applications perform file I/O that does not have such characteristics. As the degree of concurrency grows as discussed above, such random unaligned I/O will become even more prevalent, since it will become infeasible to require huge numbers of independent threads to synchronize at a point where data can be buffered into bigger segments than what each thread processes.

Finally, the managing of **metadata** (file descriptors, i-nodes, file control blocks, etc.) associated with data structures on disks is beginning to severely hamper Petascale systems. Before any I/O requests can actually touch a disk to perform the data transfers (regardless of the transfer length), the run time must determine on which disk a particular block of data in a certain file is located, whether or not some other threads are already accessing any overlapping data, and how to schedule the transfer to minimize seek time. With today's high end applications often opening literally millions of files during a single run, and with the potential for hundreds of thousands to millions of physical disks to maintain just catalogs, the amount of such metadata that must be accessed and sorted through is becoming enormous, and a severe impediment to maintaining high levels of efficiency of processing. Many of the emerging Exascale applications, especially ones maintaining massive persistent databases, will make this process even worse.

8.1.3 The Concurrency and Locality Challenge

As discussed earlier, concurrency can be measured in three ways:

- The total number of operations (such as floating point operations) that must be instantiated in each and every cycle to run the applications at Exascale speeds.
- The minimum number of threads that must be run concurrently to provide enough instructions to generate the desired operation-level concurrency.
- The overall thread-level concurrency that is needed to allow some percentage of threads to stall while performing high-latency operations, and still keep the desired dynamic thread concurrency.

8.1.3.1 Extraordinary Concurrency as the Only Game in Town

The end of increasing single compute node performance by increasing ILP (Instruction Level Parallelism) and/or higher clock rates has left explicit parallelism as the only mechanism in silicon to increase performance of a system. Thus in the embedded class, what was a single core processor will rapidly become a 1,000 core device. In the departmental scale, downsizing a Petascale machine with perhaps a large fraction of a million function units to a few racks will still require a million function units. Further, at the data center class, scaling up in absolute performance will require scaling up the number of function units required accordingly (Section 7.1.2) into the billion range.

Efficiently exploiting this level of concurrency, particularly in terms of applications programs, is a challenge for which there currently are no good solutions. Solving it requires that

- the simplicity of programming an application for a medium sized cluster of today's computers becomes as easy as programming an application today for a single core,
- the heroics needed to produce applications for today's supercomputer Petascale systems be reduced to the point where widespread departmental systems, each with different mixes of applications, are feasible,
- and that some way is found to describe efficient programs for systems where a billion separate operations must be managed at each and every clock cycle.

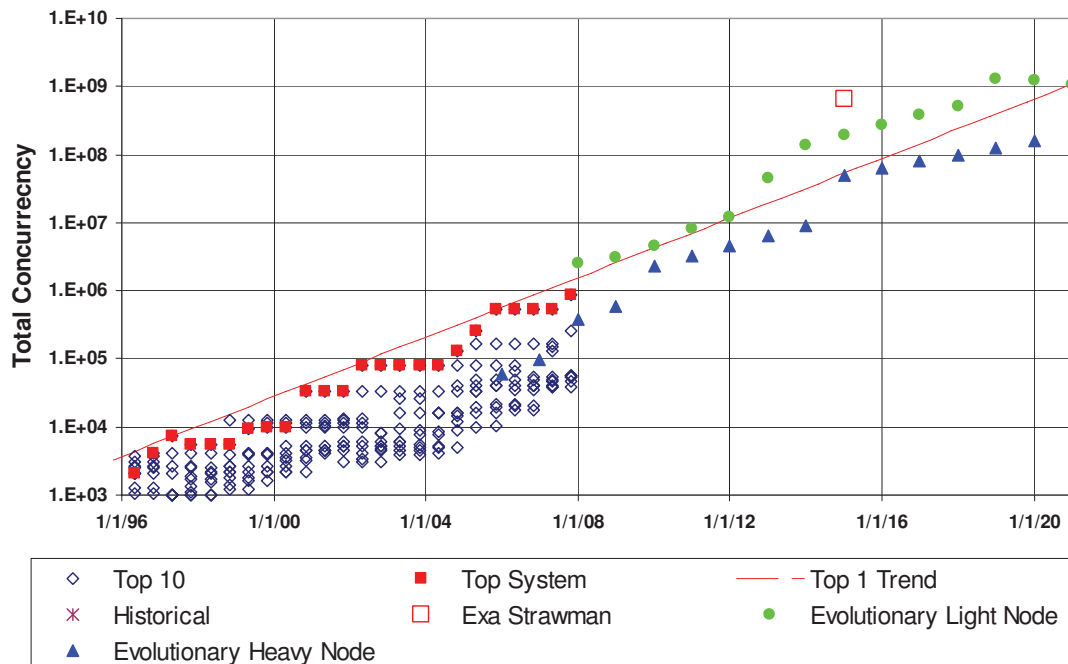


Figure 8.4: The overall concurrency challenge.

Each of these stress the state of the art beyond today's limits; especially for the data center class of systems there is little idea of even how to create "heroic" programs that actually work efficiently.

Figure 8.4 attempts to place the concurrency challenge in perspective, where concurrency is defined in Section 4.5.3 as the total number of operations (flops in this case) that must be initiated on each and every cycle. This figure is drawn by taking the trend line for concurrency from Section 4.5.3.4, and including the heavy and light node systems projection from Sections 7.2.1 and 7.2.2, the light node system projection from Section 7.2.2, and the estimate from the strawman of Section 7.3. As can be seen, even if the current trends are maintainable, billion-way concurrency will be needed for exaflops systems, and the 2015 strawman simply requires it about 5 years earlier. Further, and equally important, this level of concurrency is three orders of magnitude larger than what we have today, or expect to see near term.

Figure 8.5 graphs a similar analysis, but assuming an architecture like today's where the major unit of logic is a single-threaded processor. As discussed in Section 4.5.3.1, here there is no clean trend line that fits the top system, albeit there is a super-exponential trend in the median of the Top 10. In any case, the strawman estimate three orders of magnitude higher than any system today. Given that as the chart shows it took a decade to be able to efficiently utilize a 10X increase in processor parallelism, to expect that 1000X can be handled in less than that is a long stretch.

Making these issues of concurrency even harder is the other characteristic of the memory wall - latency. We are already at or beyond our ability to find enough activities to keep hardware busy in classical architectures while long time events such as memory references occur. While the flattening of clock rates has one positive effect in that such latencies won't get dramatically worse by themselves, the explosive growth in concurrency means that there will be substantially more

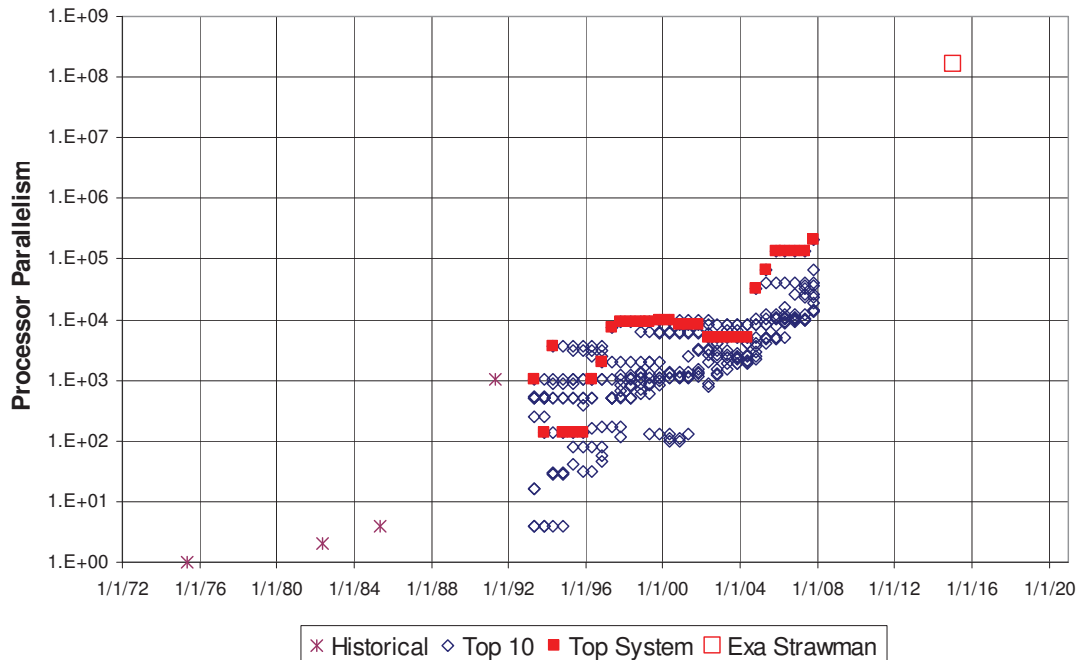


Figure 8.5: The processor parallelism challenge.

of these high latency events, and the routing, buffering, and management of all these events will introduce even more delay. When applications then require any sort of synchronization or other interaction between different threads, the effect of this latency will be to exponentially increase the facilities needed to manage independent activities, which in turn forces up the level of concurrent operations that must be derived from an application to hide them.

Further complicating this is the explosive growth in the ratio of energy to transport data versus the energy to compute with it. At the Exascale this transport energy becomes a front-and-center issue in terms of architecture. Reducing it will require creative packaging, interconnect, and architecture to make the holders for the data needed by a computation (the memory) to be energy-wise “closer to” the function units. This closeness translates directly into reducing the latency of such accesses in creative ways that are significantly better than today’s multi-level cache hierarchies.

8.1.3.2 Applications Aren’t Going in the Same Direction

Section 5.8 discussed the expected future of the scalability of applications. The summary, as pictured back in Figure 5.16, is that as we look forward both applications and the algorithms behind them seem to have some definite limits in both concurrency and locality. Overlaying on this the hardware trends as we discussed above, we get Figure 8.6, where the gap between what we expect to be able to extract from applications and what hardware as we know it today seems to be growing.

Thus a significant challenge will be in developing basic architectures, execution models, and programming models that leverage emerging packaging, signalling, and memory technologies to in fact scale to such levels of concurrency, and to do so in ways that reduce the time and energy de-

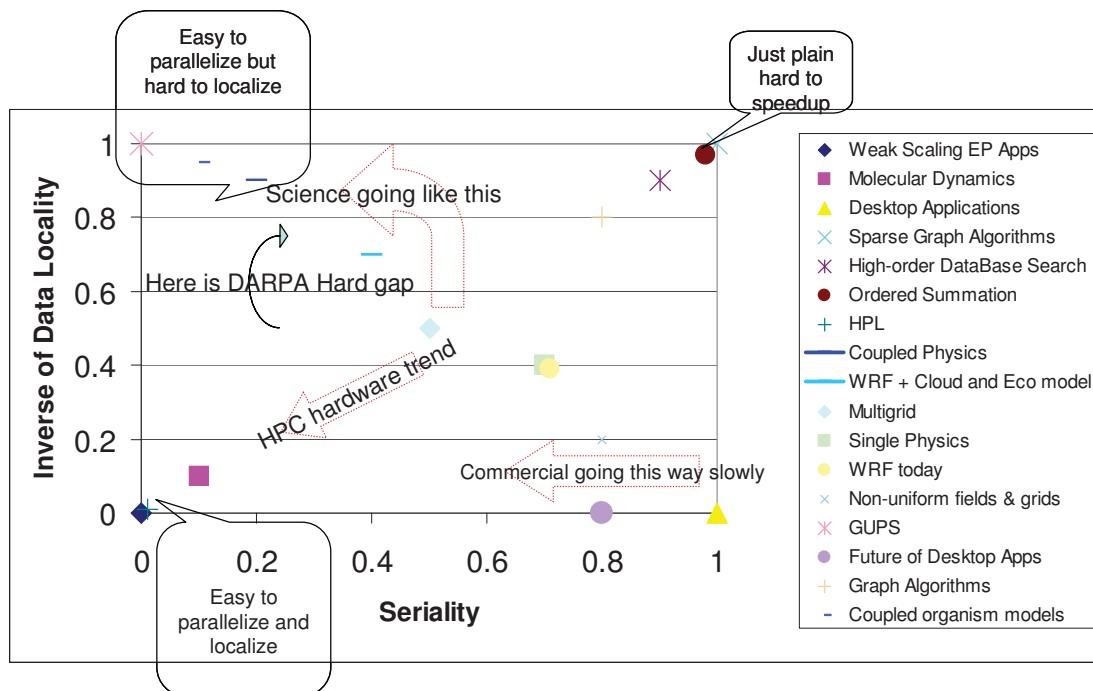


Figure 8.6: Future scaling trends present DARPA-hard challenges.

mands of access remote data in ways that applications can actually utilize the resulting concurrency in an efficient manner.

8.1.4 The Resiliency Challenge

Resiliency is the property of a system to continue effective operations even in the presence of faults either in hardware or software. The study found multiple indications that Exascale systems, especially at the data center class, will experience more and different forms of faults and disruptions than present in today's systems, including:

- Huge numbers of components, from millions to hundreds of millions of memory chips to millions of disk drives.
- Running interfaces at very high clock rates to maximize bandwidth, thus increasing both bit error rates (BER) and actual bit errors on data transmissions.
- Where leading edge silicon feature sizes are used, a growing variation in device properties across single die will increase the variation in performance characteristics of even identical circuits at different places on the die.
- In many advanced technologies there are increasing “wear-out” mechanisms in play that bring in aging effects into the fault characteristics of a device (for example, the buildup of stray charge on a gate).

- Smaller feature sizes, with less charge on a device, can in many cases increase the sensitivity of devices to single event upsets (SEU) due to cosmic rays and other radiation sources.
- Many of the technologies have significant sensitivities to temperature, making their performance characteristics a dynamic function of surrounding activity and power dissipation.
- Running silicon at lower voltages will lower power but also decreases margin, increasing the effects of noise sources such as from power supplies, and thus increasing transient errors.
- The increased levels of concurrency in a system greatly increases the number of times that different kinds of independent activity must come together at some sort of synchronization point, increasing the potential for races, metastable states, and other difficult to detect timing problems.

When taken together and placed in a system context, many of these observations tend to reinforce themselves and complicate the overall system. For example, checkpointing in high end applications requires dumping large amounts of information (in some “consistent” state) from one level of the memory hierarchy to another. How often this occurs is a function of the mean time to disruption due to a fault from which recovery is not possible. As the memory footprint of applications grow, the amount of memory that has to be fault free grows, as does the time to copy it out. The first decreases the time between checkpoints, the second increases the non-productive time to do the checkpoint, which in turn reduces the time the hardware is active, which in further turn increases the number of hardware units needed to achieve an overall performance goal, and which further increases the fault rate. Anything that an application can do to either checkpoint smaller footprints and/or indicate a willingness to ignore certain classes of errors in certain regions of code is thus clearly of huge benefit, but is currently not at all part of any application design and coding process.

Further, while many of these fault characteristics can be mitigated at design time by approaches such as ECC or duplication, not all can. As an example, variations in device parameters across a multi-core processor die results in different cores that draw different amounts of power (and thus heat differently), with different delay and thus clocking characteristics, and different sensitivities to noise, all of which may change with both local heating or aging effects. If advanced 3D chip stacking technologies are employed, then other die (with their own and different variational characteristics) will also affect and be affected by these variations. This makes the job of deciding which cores at what clock rates can be used safely a daunting real-time problem.

8.2 Research Thrust Areas

Overcoming these challenges and concerns will take a coordinated portfolio of research that is focused on some fundamental topics, but must be done within a larger context that helps direct them to Exascale-specific objectives. The study group thus looked at a whole range of topics that seemed to be of most potential impact, and grouped then into four cross-cutting **thrust areas**:

1. Co-development and optimization of Exascale Hardware Technologies and Architectures
2. Co-development and optimization of Exascale Architectures and Programming Models
3. Co-development of Exascale Algorithm, Applications, Tools, and Run-times
4. Coordinated development of Resilient Exascale Systems

This distinction between research thrust areas and challenges was deliberate; all four of the challenges are inter-related sets of problems, and solutions that address the problems represented by one challenge often affect those of other challenges. Further, even when we address the problems of just one area, solving them is not localized to a single research topic but requires co-consideration of multiple levels of the design hierarchy.

Table 8.1 overviews this relationship. The four major challenges from Section 8.1 make up the columns of Table 8.1, with the rows representing the three Thrust Areas (each discussed in detail below). Each entry in this table has two values representing criticality and gap.

Criticality is an indication by the study group as to how important effective research done in the designated research thrust is to solving the problems of the designated challenge. Thus a high value to criticality indicates the group's collective view that research in this area is absolutely crucial to solving the key problems of the challenge.

Gap is a collective view by the group of the maturity level of the current leading research in the area vis-a-vis the maturity deemed necessary to achieving Exascale systems that solve the challenge in the appropriate time frame. Thus a high value indicates the group's feeling that "business as usual" is highly unlikely to lead to viable solutions in the desired time frame.

Thus entries of the "High-High" rankings are indications that the study group believed that research into the specified areas is absolutely vital to solving the challenges, but where the current directions in research are highly unlikely to bridge the gap. These are areas where in particular additional research focus is liable to have the highest payoff.

Entries that are left blank in the Table are not areas where the group felt that there was no value to the research, only that the interaction between research and problem solution was at best indirect.

8.2.1 Thrust Area: Exascale Hardware Technologies and Architecture

In many areas it is clear that current technologies associated with the implementation of the hardware parts of systems (logic, memory, interconnect, packaging, cooling) is inadequate to solve the overall challenges, and significant research is needed. However, it is equally clear to the group that doing so in the absence of a deep understanding of how to architect such systems is liable to lead to perhaps interesting but ineffective new devices, at least for Exascale systems. Further, from experience it is also clear that today's system architectures are liable to be totally unsuited for optimizing the characteristics of such new device-level technology, and new system architectures are really needed.

Thus by grouping research topics that look at the interaction between architectures and device technologies, the group is saying that these two topics *must* be studied and developed together. Only when new device technologies are developed that blend in with system architectures that leverage their special characteristics are we liable to see overall success in solving the challenges.

As an aside, for this row in Table 8.1 an entry of "Medium" in a gap indicates that in the group's view there are laboratory demonstration devices in existence now that look promising, but that either their current path to commercialization is insufficient to get them there in time, or there is a significant lack in understanding on how to adjust Exascale architectures to leverage their properties, or (more frequently) both.

The following subsections describe several potential research topics that might fit this area.

8.2.1.1 Energy-efficient Circuits and Architecture In Silicon

Even given the promise of several non-silicon technologies, our 40 year investment in silicon means, however, that an aggressive attempt must be made to utilize silicon in some form. The challenge of building an Exascale machine (of any class) in silicon of any form that consumes an amount of power reasonable for its scale must thus be addressed at all levels. At the circuit level, there are significant opportunities to improve the energy efficiency of the basic building blocks (logic, communication, and memory) from which computing systems are built. By co-optimizing these circuits with energy-efficient architecture, even greater energy savings may be realized. Both levels of opportunities will be needed, and must be addressed together.

Most circuits and architectures today are optimized for speed, not energy. This is reflected in the supply voltage at which the circuits are operated, the threshold voltage(s) selected for the devices, the pipeline depth of the circuits, and the circuit concepts and topologies employed. We expect that substantial savings can be realized by re-optimizing all of these parameters for operations per Joule, rather than for minimum latency.

A key starting point for development of efficient circuits is in setting the power supply voltage (V_{dd}) and the device threshold voltages (V_{TN} and V_{TP}). The ITRS projections for the 32nm node project a supply voltage of 0.9V and a threshold voltage of 0.3V. Because energy scales as V^2 , significant energy savings can be realized by reducing the supply voltage. The strawman of Section 7.3 reduces operation energy by a factor of nearly three by reducing V_{DD} to 0.6V. Some preliminary studies (see Section 6.2.2.3) suggest that for static CMOS logic, the supply voltage that optimizes operations per Joule is just slightly above the threshold voltage (320mV for $V_T = 300$ mV see Figure 6.10, repeated here as Figure 8.7).

The optimization is more involved, however, because threshold voltage is also a free variable. As the supply voltage is reduced, threshold voltage should also be reduced to the point where leakage power and dynamic power are balanced. The optimum setting of V_{dd} and V_T is also dependent on the circuit being optimized and its duty factor. The activity factor largely drives the balance of dynamic and static power. Also, some efficient circuit forms require some headroom - requiring higher supply voltages. Moreover a given circuit is not restricted to a single supply voltage or a single threshold voltage. Multiple threshold voltages are already used to advantage in modern circuits and multiple supply voltages may be employed as well. A study of energy efficient circuits needs to start with a careful examination of supply and threshold voltages in the context of an Exascale system.

The circuits employed in modern computer systems can be roughly broken down into those used for communication, memory, and logic. We discuss each of these briefly.

Communication circuits move bits from one location to another in a system with different circuits being employed at different levels of the packaging hierarchy. Static CMOS drivers and repeaters are typically used to drive on-chip signals, while high-speed SerDes are often used to drive signals between chips.

The circuits used to transmit global on-chip signals are a great example of the potential energy savings from optimized circuits. On-chip signal lines have a capacitance of about 300fF/mm. With conventional full-swing speed-optimal repeaters, the repeater capacitance equals the line capacitance for a total of 600fF/mm. At the ITRS supply level of 0.9V, sending a bit on chip using conventional full-swing signaling requires about 0.5pJ/mm.

If we optimize on-chip transmission for energy rather than speed, we can significantly reduce the energy required to transport them. First, we reduce the signal swing V_S to a level which minimizes energy/bit. Note that this is not the minimum possible V_S , but rather the level which balances transmit energy (that reduces with V_S) against receive energy (which increases as V_S is reduced).

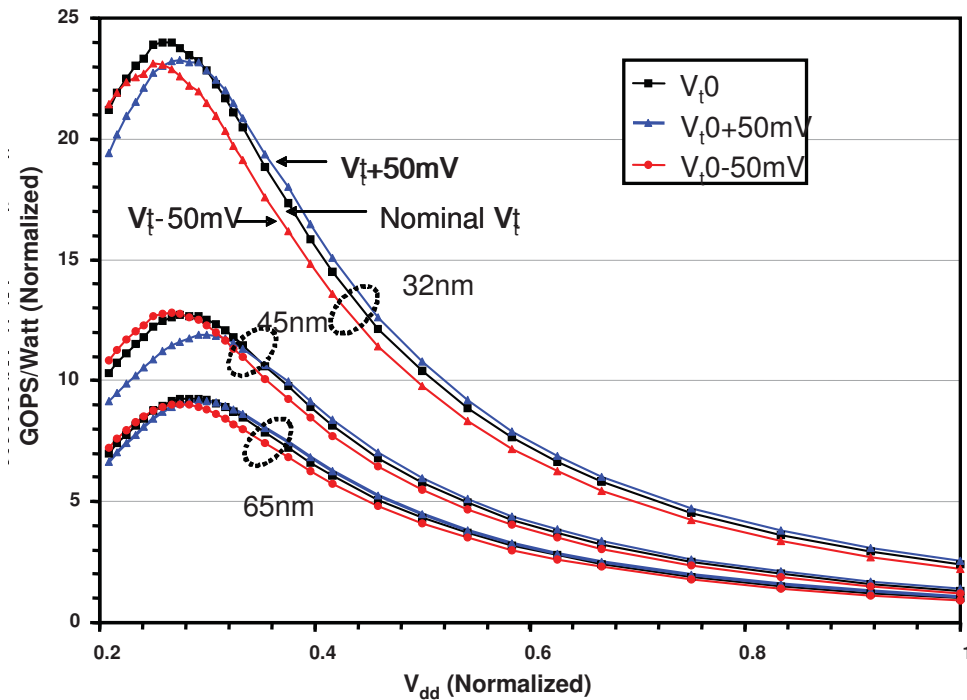


Figure 8.7: Sensitivities to changing V_{dd} .

With a V_S of 100mV, for example, generated from an efficient switching supply, the transmission energy would be about 6fJ/mm, a savings of nearly two orders of magnitude.¹ Optimizing repeater spacing, line geometry, and equalization for energy may also yield significant gains.

The serializer/deserializer (SerDes) circuits that are used for high-speed off-chip communication have energy that is dominated by timing, serialization, and deserialization. Over 2/3 of the power consumed by a recent energy-efficient SerDes was consumed by these functions - with the remaining 1/3 used by the actual transmitter and receiver. While the gains expected for off-chip communication are more modest than for on-chip, efficiency improvements of one order of magnitude or more are possible from improvements in these circuits and the protocols they implement.

It is clear that optimized communication circuits have the potential to reduce signaling energy by as much as two orders of magnitude. Such a dramatic change in communication cost changes the space of architectures that are possible, and greatly changes which architectures are optimal. Hence any architecture optimization must take optimized signaling into account.

Memory circuits move bits forward in time - storing data and later retrieving it. On the processor die memories are typically implemented today using 6-transistor SRAM circuits, while the bulk memory is usually implemented as 1-transistor DRAM. There is the potential for significant energy improvement in both. For the on-chip SRAM, access energy is dominated by charging and discharging bit lines. This can be reduced by either reducing voltage swing or by employing hierarchical bit lines to reduce the amount of capacitance switched. For the bulk DRAMs, bounding arguments suggest that one should be able to access a bit of DRAM for about 3pJ of energy, yet for conventional DRAMs access energy is about 60pJ/bit.

¹The straw man of Section 7.3 assumes a less aggressive 20fJ/mm per bit.

As with communication circuits, optimized memory circuits can result in order-of-magnitude reductions in access energy which will have a substantial impact on machine architecture.

Logic circuits perform arithmetic and control the operation of a computer system. There are fewer opportunities to improve the energy of silicon logic circuits because static CMOS circuits are already highly efficient and designs for common arithmetic functions are already highly optimized. Gains may be realized by optimizing supply and threshold voltages. Additional gains may be realized through machine organization - e.g. using shallow pipelines and arithmetic unit organizations that reduce energy.

8.2.1.2 Alternative Low-energy Devices and Circuits for Logic and Memory

Given the criticality of driving energy per operation down, and the uncertainty surrounding silicon, the study group believes that it is equally important to develop an understanding of the potential for alternative device technologies to yield suitable low-energy high-performance circuits for both computational functions and storage. Equally important is sufficient density to meet or exceed that possible with traditional technologies.

The group believes that several such technologies may already exist, at different levels of maturity. RSFQ (Rapid Single Flux Quantum) devices have been demonstrated in several projects such as HTMT[140] in some real circuits at huge clock rates and very low power, but not in system architectures that deliver large memory capacities.

Additionally, several labs have begun discussing crossbar architectures from novel bi-state devices (see HP's SNIC[137] and IBM's SCM[61]) that can either provide very dense memory, or a dense new form of combinational logic with minimal active devices, or enable a new generation of reprogrammable logic that can work in conjunction with silicon devices underneath to reduce power by dynamic circuit reconfiguration. Again, however, what is lacking is a careful study of how best to utilize such devices in Exascale circuits to minimize total energy per operation while still having sufficient performance not to make the concurrency and capacity problems worse.

8.2.1.3 Alternative Low-energy Systems for Memory and Storage

As shown in the strawman of Section 7.3, power in the memory system is a severe problem. This power comes in several forms:

1. maintaining the actual storage arrays themselves, in whatever technology they are implemented.
2. accessing these arrays, and extracting the desired information.
3. maintaining "copies" of such data in other memories with better locality than their home.
4. moving data from arrays to off-chip (for memory-chip technologies)
5. moving data between chips, boards, and racks.
6. implementing the storage access protocols needed to trigger the transfer and synchronization of such data.

Clearly there are issues and tradeoffs to be made here in terms of density of storage medium and bandwidth, just as we have wrestled with for years. These get only dramatically worse when we attempt to use today's technologies and memory architectures in Exascale systems, especially the data center class.

While there is promise from several emerging technologies to address some of these issues, the only way to address the whole problem in a more than incremental way is to refocus in a serious fashion the whole notion of a memory hierarchy. The capabilities of some of the emerging technologies need to be blended with new architectural ideas that truly do leverage them in a way where they integrate with the other parts of the system to dramatically reduce energy. Section 7.5.4 is one example of an attempt to begin the exploration of such a space. In general, a complete exploration will require simultaneous attention to multiple dimensions, including:

- Considering new levels in the memory hierarchy where the characteristics of emerging or alternative memory technologies can bridge the gap between, for example, DRAM and disk storage, especially for some critical functions such as checkpointing and metadata management.
- Explicitly and pro-actively managing data movement between levels of the memory hierarchy to reduce power consumed by such movement.
- Rearchitecting conventional DRAMs in ways that greatly reduce the energy per bit access when used in massive memory arrays where high concurrency of access and bandwidth are required.
- Developing multi-chip and 3D packaging that allows chips of either of the above types to be efficiently stacked in ways that minimize the *total energy cost* of access, while still retaining the high access rates needed.
- Developing alternative memory access paths and protocols using photonics for example, that actually provide end-to-end reductions in both energy per bit accessed and time per access.
- Mixing and matching the right suite of storage devices, transport media, and protocols with the careful positioning of function units to minimize not only unnecessary copies, but also the total power needed to execute latency-sensitive pieces of code.

8.2.1.4 3D Interconnect, Packaging, and Cooling

The degree of difficulty posed by the packaging challenge depends on the memory bandwidth requirements of the eventual computer, as discussed in Section 7.3, and abstracted in Section 6.5. The lower end of the requirement, 44 GBps from the CPU to each of 16 DRAM die can most likely be satisfied with conventional high-end packaging. The high-end requirement of 320 GBps to each of 32 DRAMs is well beyond the means of any currently available packaging technology. Some embedded applications were investigated to obtain another perspective on the possible size requirement. As discussed elsewhere in this document, the usual requirement of 1 Byte per second and 1 Byte of memory for every FLOPS, would require larger scales of memory system but would have significant power issues. A system anywhere near this scale would require significant advances in interconnect and packaging. In particular, advances in 3D system geometries need to be considered.

Advances in 3D packaging also presents an opportunity to use geometry to reduce power consumption. With a conventional packaging approach, aggressive scaling of interconnect power would permit memory-CPU communications at an energy cost of around 2 pJ/bit. On the other hand, some 3D integration technologies, as discussed in Section 7.1.5 may permit power levels to approach 1-20 fJ/bit, depending on the length of run of on-chip interconnect required. Even at the low end bandwidth of 16 x 44 GBps, this represents a potential power savings of around 10 W per module, which could be more productively used for FPUs or additional memory capacity.

As well as provisioning interconnect, packaging also plays roles in power and ground current distribution, noise control (through embedded capacitors), heat removal and mechanical support, so as to ensure high reliability. The simultaneous roles of current delivery and heat removal create a geometric conundrum as the high power areas of the chip need lots of both at the same time. Again, this is a case where a concerted effort that involves both architecture and technology expertise is needed to arrive at practical solutions that would have real effect on achieving Exascale metrics.

Finally, it should be realized that advances in areas outside of packaging could simplify the creation of an Exascale 3D solution. In particular, if efficient voltage conversion could be incorporated within a 3D chip stack, then the “two-sided problem” is greatly simplified. Delivering 100 A at 1 V is a lot harder than delivering 1 A at 100 V. Similarly, releasing one side from a cooling function provides a similar improvement. For example, incorporating silicon micro-channel cooling into the chip stack removes the need for using one side for heat removal. Again, this is a difficult co-optimization problem if a truly useable solution is to be found.

On the topic of cooling, it might be that an initial embedded Exascale computing stack would be air-cooled. Large scale deployment of water chillers and circulating coolants, such as feasible for the data center or even departmental classes, do not lend themselves to embedded solutions in war fighting craft and vehicles. However, this global issues does not prevent designs from using liquid phase solutions locally (inside the stack). As long as the liquid does not require replenishment, then such local solutions might be reasonable. There are severe challenges in such solutions though. Leaks are still a problem in liquid cooling, and a leak-free technology would be a worthwhile investment. Also, techniques to circulate and pump the coolant are needed on the physical scale of a small 3D package. Similarly, local heat exchangers would be needed if local liquid cooling solutions are to be used. Heat pipes provide an excellent way to use liquid phase cooling locally without mechanical complexity. Thus, advances in the capacity of thin heat-pipe like solutions would be welcome in an Exascale computer.

8.2.1.5 Photonic Interconnect Research Opportunities and Goals

Photonics for long-haul data communication has long been believed to have real potential for significant power and latency reduction, especially when very high bandwidths are needed (see Section 7.5). Recent research (see for example Section 7.5.4) has opened up some tantalizing alternatives that push photonics far deeper into a system architecture than considered in the past. Given the pressing need to reduce energy and power throughout an Exascale system, it is incumbent that a complete research program fully explore the potentials for such technologies for their possible infusion, and do so in a way where the potential advantages can be fully compared on a common basis to the best of conventional wire based communication.

Further, as with many of the alternative technologies discussed earlier, it is also essential that such studies tightly integrate both architectural, circuit-level, and technological innovation and tradeoffs, and do so with metrics that cover not just power and bandwidth but also ones that are relevant to the other major challenges discussed in this report, such as resiliency. Consequently, a well-designed program in this area will need to focus on both long and short range communication. To a large extent, the long-range studies are already under way in industry, but efforts must be made to ensure their relevance to both data center and departmental class Exascale peta systems.

The goal of “short-range” photonic studies need to determine whether or not there are in fact gains to be made by creative use of new optical devices, with the key goal of establishing on a true “apples-to-apples” basis what the energy and performance gains really are possible. As before, this can only be done by considering microarchitectural alternatives concurrently with alternatives in interconnect devices and protocols for the photonic-digital interface. This will involve modeling for

performance, power, and resiliency.

8.2.2 Thrust Area: Exascale Architectures and Programming Models

Just as slowing clock and memory density growth have driven the need for alternative technologies and architectures which can incorporate them, the expected explosive growth in concurrency is driving the need to explore and develop better ways to connect what is programmed with what is left visible to programs from the underlying architectures. Thus the group feels that a second major research thrust must involve new technologies to bridge the gap, and again do so with a strong component of architectural development.

The following subsections describe several potential research topics that might fit this area.

8.2.2.1 Systems Architectures and Programming Models to Reduce Communication

The toughest of the power problems is not on circuits for computation, but on communication. Communication is insidious - a pJ wasted on moving data is not only a pJ that is not available for communication, but also a pJ that must often be matched by other pJs that must be expended to monitor the progress of the data during the movement, to correct for any errors, and to manage the suspension and reawakening of any circuits that source or sink the data.

Further, simply minimizing communication power alone may not minimize overall power. If higher power signalling can reduce latency, then perhaps more energy may be saved in maintaining the state of higher level threads, which in turn may reduce the number of threads that must be maintained in a register file, which in turn may allow the size of the register file to become smaller, which in turn may reduce the energy associated with all accesses to that file.

Hence there is a real need for the development of architectures and matching programming systems solutions that focus on reducing communications, and thus communication power. Such solutions would allow more of the total power to be used for computation.

Examples of potential techniques include:

- Over-provision the design with both function units and communications paths, but with flexible activation so as we can either use all available power in the function units if the problem is computation limited and all data is local, or use all the power for communication if the problem is global communication limited.
- Design in self-awareness of the status of energy usage at all levels, and the ability to adjust the activation of either logic or communication circuits to maintain a specific power level below a maximum while maintaining specified performance or system throughput goals.
- Provide more explicit and dynamic program control over the contents of memory structures closest to the function units so that minimal communication energy can be expended.
- Develop alternative execution and programming models where short sequences of code can be exported to data that is not resident, rather than dragging the data all across the system.

8.2.2.2 Locality-aware Architectures

The energy required to access an operand or instruction increases significantly with distance, with a particularly large increase occurring when moving from on-chip to off-chip. Hence data locality is a key element of reducing energy per operation to a level that enables Exascale systems. To operate efficiently, the bulk of data accesses must come from small registers and memories co-located with

the arithmetic units consuming the data. The number of high-energy off-chip and cross-machine accesses must be limited.

The problem of data locality spans several abstraction levels, and achieving high data locality requires a **locality-aware** architecture, a programming system that optimizes data placement and movement, and applications with inherent locality. Some applications (and portions of applications) fundamentally require large numbers of global accesses and will not benefit from locality-aware architectures and programming systems. For other applications, however, large gains can be realized by keeping data local and explicitly controlling its movement.

In this section we discuss a possible thrust to develop locality-aware architectures. Almost all architectures employ a storage hierarchy of some variety. These hierarchies are distinguished by their structure (the sizes of memory arrays and how they are connected), their management (implicit and/or explicit mapping and replacement policies), and how power is budgeted across the levels of the hierarchy.

The structure of the storage hierarchy limits what management and power policies are possible, since these policies operate over the structure. Any structure includes at least a level of registers close to the arithmetic units, one or more levels of on-chip storage arrays interconnected by a network, a level of per-chip or per-node off-chip storage, and one or more levels of global storage, also interconnected by a network. The granularity of the storage levels is often driven by efficient sizes for storage arrays and the levels are often designed to fit the packaging hierarchy. As machines increase in scale - both more cores per chip and more chips - we expect the number of levels and the complexity of certain levels to increase.

Most contemporary machines manage the hierarchy between registers and main memory *implicitly* (i.e.: as caches) and *reactively* (moving data in response to a miss at one level in the hierarchy) and use a *uniform* mapping at each level. While conceptually simple, these management policies leave large opportunities for improvement in energy efficiency and performance. Managing L2 or L3 on-chip storage uniformly, for example, results in distributing data across the entire area of the storage level - often the entire chip. Using non-uniform mapping policies can potentially reduce communication (and hence energy) by placing data nearer to its point of use.

Implicit and reactive caches wait to fetch data until it is requested - which increases the amount of concurrency required to cover the resulting long latency - and replace the data according to a simple least-recently-used policy. Managing the same memory structures in an explicit and proactive manner (i.e., explicitly transferring data from one level to another before it is needed, as in the IBM Cell) has been shown to yield large improvements in both efficiency (by eliminating unneeded data movement) and performance (by overlapping load latency with operations). The fully-associative nature of an explicitly-managed memory hierarchy means that arbitrary data can be placed in a memory structure, limited only by the capacity of the memory. Software can thus count on deterministic access latencies, and can be scheduled by the compiler to achieve high utilization. Explicit management is particularly attractive when combined with locality-enhancing program transformations so that a fragment of a data set can be produced in a local memory and consumed from the same memory without ever having to be read from or be written to main memory.

Explicit management, however, poses programming challenges for irregular applications that share mutable data. Copying data into a local memory with a separate address space can lead to incoherent data updates if proper synchronization is not employed. A hybrid management strategy that explicitly moves data while providing the needed exclusion and synchronization holds the potential to provide performance and efficiency gains even for irregular codes with mutable data. Research on hardware/software approaches for structuring and efficiently managing a storage hierarchy is likely to yield large gains in efficiency while at the same time reducing the required

concurrency and simplifying programming.

Explicit management poses other challenges for the compiler. Runtime data dependencies may make it difficult or impossible to know where a given piece of data is located. Furthermore, legacy applications may need extensive, whole-program-level transformations affecting the order in which data is traversed in order to exploit a locality-aware machine architecture. For this reason, a locality aware architecture may require (or at least be best exploited by) a locality-aware programming model. Research in compiler technologies and/or programming models for locality-aware architectures should be focused on achieving high locality without burdening the programmer with detailed control of data movement.

Power is the critical resource in emerging computer systems and a memory hierarchy is largely defined by how power is budgeted across the levels of the hierarchy. Putting the bulk of the power into arithmetic units, registers, and L1 access, for example, gives a machine that performs well on highly local problems at the expense of problems dominated by global accesses. A machine that puts the bulk of its power into global accesses is optimized in the opposite direction.

Because modern chips are power, not area, constrained, it is possible to build a multi-core computing node that can spend its entire power budget on any one level of the hierarchy. Such a *power-adaptive architecture* would be able to use its total power budget across a wide range of program locality - rather than being constrained by a fixed power allocation. Such an architecture, however, requires power management hardware and software to ensure that it does not exceed its total power budget in a way that would either collapse the power supply or result in an over-temperature situation. Research in power-aware architectures and management policies will not improve energy efficiency, but rather will allow a machine with a fixed power limit achieve a larger fraction of its peak performance by adapting its power budget to the needs of the application.

8.2.3 Thrust Area: Exascale Algorithm and Application Development

In the past two decades there has been a three order-of-magnitude growth in the amount of concurrency that application and algorithm developers must confront. Where once it was $O(100)$ operations in a vector, it is now $O(100,000)$ individual processors, each furnished with multiple floating-point arithmetic processing units (ALUs). This dramatic increase in concurrency has lead the dramatic lessening of efficient application scaling. Small load imbalances or synchronization events can create an Amdahl fraction that prevents most of today's applications from efficiently utilizing hundreds of thousands of processors.

As we look forward to Exascale computing, using processors whose clock frequencies will be at best marginally higher than those in use today, we anticipate a further growth in ALU count of *another* three orders-of-magnitude. On top of this will likely be another one or two orders-of-magnitude increase in the number of *threads* as processor vendors increasingly turn to multi-threading as a strategy for tolerating the latency of main memory accesses (i.e., cache misses). The challenge of developing applications that can effectively express $O(10^{10})$ threads of concurrent operations will be daunting. Equally difficult will be avoiding even the smallest unnecessary synchronization overheads, load imbalances, or accesses to remote data.

In addition, historically, software developers have striven to minimize the number of operations performed. More recently, as the memory hierarchies of today's mainstream systems have made main memory references increasingly costly, the emphasis has turned to organizing the operations so as to keep them in cache. At Exascale, where fetching a word from DRAM will cost more power than performing a double precision floating point multiply-accumulate operation with it, there will be a new imperative to minimize state size and eliminate unnecessary references to deep in the memory hierarchy. Research into a new generation of algorithms which repeat calculations to avoid

storing intermediate values or use additional calculations to compress state size will be needed.

Similar and equally vexing difficulties will arise when trying to counter the other challenges of resiliency, locality, and storage capacity.

Together, this calls for a substantial research effort to fund a variety of areas such as discussed in the following subsections.

8.2.3.1 Power and Resiliency Models in Application Models

Going forward it will be critical to understand the impacts of architectural choices on the performance and reliability of applications. For example, one will wish to measure or predict the “science results per watt” of a specific design in terms of intended uses. If an algorithm or application can be made less compute-intensive it may require fewer, or slower/cooler functional units. On the other hand if it is already memory bandwidth-bound at Petascale we want to know that and perhaps invest more of the energy budget in the memory subsystem. Related to energy and memory, if an algorithm’s or application’s overall memory footprint or locality clusters can be improved, then perhaps a power-aware architecture can run it for less energy. And related to concurrency, one must understand “what is the inherent concurrency of an algorithm or application?” to be able to say what sort of processors, and how many processors, will enable it to run at Exascale. Finally, as to resiliency, we must understand the fault-tolerance and checkpointing overhead of algorithms and applications to be able to say if they can run at all on proposed architectures, or if they require more development work to make them resilient enough to survive the anticipated hardware fault rates.

An initial research thrust then is simply to characterize the computational intensity, locality, memory footprints and access patterns, communications patterns, and resiliency of existing nearly-Petascale applications deemed most likely to go to Exascale, create performance models of these, and connect these performance models to power and failure-rate models from the Exascale technologists and architects. This will tell us how much power these applications will consume, how fast they will run, how often they will fail etc. at various system design points. We want to construct unified application-system models that enable critical-path analysis, thus to say where the energy budget is being expended, what are the performance bottlenecks, where will irrecoverable failures happen first, on a per-application basis.

8.2.3.2 Understanding and Adapting Old Algorithms

It is probably the case that few if any of today’s applications would run at Exascale. Reasons might include: inability to scale to the new levels of concurrency, inefficient performance at whatever levels it can reach, poor energy per operation management, or inadequate checkpointing considerations that do not match the resiliency characteristics of the machine.

In most cases, we are not going to replace fundamental mathematical kernels. Therefore we will be forced to learn how to adapt them to this extraordinary scale. This will likely involve reformulating such algorithms as well as expressing them in new programming languages. An example might be to try to reformulate Krylov space algorithms so as to minimize the number of dot-products (and hence the global synchronization necessary to sum up across huge processor sections) without a subsequent loss in numerical robustness.

In addition, we will also have to learn to tune and scale these algorithms for the target machines, most probably using whatever modeling facilities might become available as discussed in the prior section. For example, it may be that applications inherently have ample fine-grained asynchronous parallelism, but that it is poorly expressed in the inherently course-grained and synchronous MPI

of today. Models will tell us the inherent parallelism and thus predict the potential suitability and benefit to converting the application to modern, more expressive and asynchronous, programming paradigms and languages.

As to locality, many applications may have inherent locality that is poorly expressed, or impossible to exploit with today's clumsy cache and prefetch management systems, or is present but ill-understood. As mentioned in Chapter 5, tools should be developed as part of this research thrust to help identify inherent locality, while programming languages should allow hooks to control the memory hierarchy of power-aware machines to exploit locality. As to checkpointing, it may be that crude checkpointing (write out entire memory image every time-step) works acceptably at Petascale but is impossible to do because of size and overheads of data at Exascale. Again, an algorithm reformulation approach that uses models can indicate this and guide development of refined smarter and cheaper application specific checkpointing strategies.

8.2.3.3 Inventing New Algorithms

Developing new algorithms that adapt to Exascale systems will involve more than just finding algorithms with sufficient amounts of useable concurrency. These systems will be extremely sensitive to starvation, and will require a new generation of graph-partitioning heuristics and dynamic load-balancing techniques that need to be developed and integrated hand-in-hand with the basic computational kernels. There will be a premium on locality to minimize unnecessary data movement and the concomitant expense of both power and latency. Exascale systems will likely have a much higher ratio of computing power over memory volume than today's systems, and this will open up room for new algorithms that exploit this surfeit of processing power to reduce their memory size.

8.2.3.4 Inventing New Applications

Applications are typically complex compositions of multiple algorithms interacting through a variety of data structures, and thus are entities separate from the underlying algorithms discussed above. Exascale systems cannot be designed independently of the target applications that will run on them, lest there be a vanishingly small number of them. It has taken a decade for applications to evolve to the point where they express enough concurrency to exploit today's trans-Petascale systems. It will be harder still for them to stretch to Exascale. Clearly it will also be important to develop a keen understanding of how to develop full-scale *applications* suitable for execution on Exascale systems. For example, future DoE applications will most likely be focused on developing massively parallel coupled-physics calculations. While these science drivers are inherently billion-way parallel they risk being less efficient, not more efficient, than today's mono-physics codes would be at Exascale, for reasons elaborated in Section 5. And, as shown in Figure 8.6 the trend may be away from easy-to-exploit locality in these applications. An important research thrust then will be to invest in scalable algorithm and application development that preserves locality. However an emphasis of this research thrust might again be to provide parameterized power/reliability/performance models to application development teams so they can proceed with such development in parallel with Exascale hardware and system research, and be ready to take advantage of new architectures when they emerge. Again the search for locality in applications is symbiotic with the ability of power-aware architectures to manage locality explicitly (cache only the things I want cached, don't prefetch the things I don't want prefetched etc.) It is important though that Exascale architecture not be carried out in a vacuum but inform and be informed by development of Exascale applications. Thus we propose a specific research thrust in this area towards developing models that can be a

Lingua Franca for communicating to applications developers the capabilities, limitations, and best usage models, for Exascale technologies and architectures.

8.2.3.5 Making Applications Resiliency-Aware

Finally, the development of **Resiliency-aware applications** will become critically valuable to achieving genuinely useful Exascale. We must understand the fault-tolerance and checkpointing overhead of algorithms and applications to be able to say if they can run at all on proposed architectures or if they require more development work to make them resilient enough to survive the anticipated hardware fault rates. Applications may need to specify where higher guarantees of correctness are necessary, and not necessary, in order to avoid such techniques as brute duplication that can rapidly run up power consumption.

8.2.4 Thrust Area: Resilient Exascale Systems

Technology trends and increased device count pose fundamental challenges in resisting intermittent hardware and software errors, device degradation and wearout, and device variability. Unfortunately, traditional resiliency solutions will not be sufficient. Extensive hardware redundancy could dramatically improve resiliency, but at a cost of 2-3 times more power. Checkpointing is effective at providing fault recovery, but when error rates become sufficiently high, the system may need to be saving checkpoints all of the time, driving down the time left for useful computation. To increase system resiliency without excess power or performance costs, we recommend a multi-pronged approach which includes innovations at each level of the system hierarchy and a vertically integrated effort that enables new resiliency strategies across the levels.

8.2.4.1 Energy-efficient Error Detection and Correction Architectures

The tradeoff between resiliency and power consumption manifests itself in several ways. Reducing power supply voltages toward the threshold voltage reduces power consumption in the primary circuits but may require more rigorous resiliency to handle higher error rates. In memories, shortening the refresh rate reduces power consumption but may increase the bit-error rate. Error rates can also be a function of the application characteristics or the temperature. We recommend research into low overhead hardware mechanisms to detect (and potentially correct) errors at the logic block level, rather than the gate level. Past research into arithmetic residual computation checking and on-line control logic protocol verification are starting points for this work. We also recommend tunable resiliency in which the degree of resiliency (and therefore the power required for it) can vary depending on the demand. Examples include selective redundant execution in software through re-execution or in hardware through on-the-fly processor pairing.

8.2.4.2 Fail-in-place and Self-Healing Systems

Because Exascale systems will experience frequent component failures, they must be able to continue operating at peak or near-peak capacity without requiring constant service. Research into fail-in-place and self-healing systems will be necessary to achieve this goal. Exploiting the redundancy inherent in Exascale systems in a manner transparent to application programmers will be critical. Hardware redundancy may include spare cores per chip, spare chips per node, memory modules, or path diversity in the interconnection network. The key to exploiting the redundancy transparently will likely be virtualization so that applications need not care precisely which resources are being used. Redundancy and virtualization techniques will be necessary to tolerate manufacturing defects,

device variation, aging, and degradation. The system must be able to reconfigure its hardware and software to make the system resilient to frequent hardware failures. This goal also reaches up to the packaging design so that system modules can be easily hot-swapped without halting the system. Emerging nanoscale devices, such as hybrid CMOS Field Programmable Nanowire Interconnect architectures, will require redundancy and automatic reconfiguration just to tolerate the inherent unpredictability in the manufacturing processes.

8.2.4.3 Checkpoint Rollback and Recovery

The dominant form of system recovery relies on checkpoint-restart and would benefit from innovations to reduce checkpointing overheads. For example, using solid-state non-volatile level of the memory hierarchy (flash memory) could provide substantial bandwidth for checkpointing so that it could be done under hardware control, drastically reducing checkpoint latency. Another avenue for research is in intelligent checkpointing, at the application or system levels. Such intelligent checkpointing schemes would select when and which data structures to checkpoint so that no unnecessary data is saved or time is wasted. Ideally, compilers or run-time systems would automatically select when and what to checkpoint, but an intermediate approach that runs semi-automatically (perhaps relying on application-level directives) would be a substantial advance over the state-of-the-art. Another approach is to shift to a programming model that is inherently amenable to checkpointing and recovery, such as transactions from the database and commercial computing domains. Finally, the community should prepare itself for a time when parallel systems must be continuously checkpointing due to the high error rates. Providing resiliency to such systems will require a range of solutions at the software system level.

8.2.4.4 Algorithmic-level Fault Checking and Fault Resiliency

Algorithmic based fault tolerance can provide extremely efficient error detection as only a small amount of computation is required to check the results for a large amount of computation. Prior research has produced algorithmic approaches for matrix multiply, QR factorization, FFT, and multi-grid methods. Further research in this area could expand the domain of algorithms suitable for efficient checking. Additional research will also be needed to automatically provide error checking for intermediate results in these algorithms, as high error rates may prevent any long-running computation from completing without correction. Another class of applications that can provide efficient correction are those that are self-healing, in which the algorithm can automatically correct errors. Some convergence algorithms already have this property as some data errors in one iteration can be corrected in subsequent iterations. Finally, applications that require less precision than the computer numerical representations provide may not care if errors exist in insignificant bits. The rise in error rates may require application and software tools writers to optimize for fast error detection and recovery in order to make good use of the inherently unreliable hardware.

8.2.4.5 Vertically-Integrated Resilient Systems

A vertically integrated approach will span all levels of the system stack including applications, programming systems, operating systems, and hardware. Achieving a system that is optimized as a whole for resiliency will require substantial research at the system level, but can result higher overall resiliency at a lower cost. As an example, a vertically integrated system might employ watchdog timers to detect faults in control logic, but use programming system controlled checking threads to provide redundancy for the critical portions of an application. Selective replication eliminates duplicate hardware and is applied only when necessary. Checkpointing would still be

required for recovery from control or data errors, but intelligent checkpointing (at the application or system level) in conjunction with fast hardware checkpointing mechanisms could reduce the overhead experienced by the application. System-level resiliency can be an enabling technology by allowing less reliable or new devices to be manufactured at lower cost; such techniques may be necessary for continued device scaling or to bridge the gap beyond traditional semiconductor technologies.

8.3 Multi-phase Technology Development

Given the relative immaturity of many of the technologies mentioned above as potential members of an ultimate Exascale technology portfolio, it made sense to the study group that bringing them to a level of technological maturity commensurate with a 2015 timeframe would require a careful phasing of the research efforts into roughly three phases:

1. System architecture explorations
2. Technology demonstrations
3. Scalability slice prototyping

The most immature of the technology solutions will have to go through all three, while others than have, for example, lab demonstrations, may simply have to accelerate their transition through the last two.

8.3.1 Phase 1: Systems Architecture Explorations

This would be the earliest phase for those challenges where the level of technological maturity is very low. The goal for research in this phase is to propose potential new alternatives, develop a coherent understanding of the interaction of the new technologies with architectures that will scale to the exa level, and identify what are the critical experiments that need to be performed to verify their applicability.

8.3.2 Phase 2: Technology Demonstrators

Research at this phase would focus on demonstrating solutions to the “long poles” of challenges using new technologies developed in the first phase. Each such demonstration would most probably be done in isolation of integration with other new technologies, so that the real properties and attributes of a single technology can be identified and measured with a fair degree of confidence.

The purpose here is not to demonstrate system solutions or even subsystems, but to develop and demonstrate targeted pieces of new technology in ways that would provide system implementers enough confidence that they could start planning for its introduction into Exascale products.

8.3.3 Phase 3: Scalability Slice Prototype

This phase would build on multiple demonstrations from the second phase to integrate different pieces of technology in ways that represent a relatively complete end-to-end “slice” through a potential Exascale system. This slice is not expected to be a complete system, but should include enough of multiple subsystems from a potential real system that the scaling to a real, complete, system integration is now feasible and believable.

Such a phase is an absolute necessity were significant deviations from existing architectures, programming models, and tools are forecast.

This page intentionally left blank.

Appendix A

Exascale Study Group Members

A.1 Committee Members

Academia and Industry	
Name	Organization
Peter M. Kogge, Chair	University of Notre Dame
Keren Bergman	Columbia University
Shekhar Borkar	Intel Corporation
William W. Carlson	Institute for Defense Analysis
William J. Dally	Stanford University
Monty Denneau	IBM Corporation
Paul D. Franzone	North Carolina State University
Stephen W. Keckler	University of Texas at Austin
Dean Klein	Micron Technology
Robert F. Lucas	University of Southern California Information Sciences Institute
Steve Scott	Cray, Inc.
Allan E. Snaveley	San Diego Supercomputer Center
Thomas L. Sterling	Louisiana State University
R. Stanley Williams	Hewlett-Packard Laboratories
Katherine A. Yelick	University of California at Berkeley
Government and Support	
William Harrod, Organizer	Defense Advanced Research Projects Agency
Daniel P. Campbell	Georgia Institute of Technology
Kerry L. Hill	Air Force Research Laboratory
Jon C. Hiller	Science & Technology Associates
Sherman Karp	Consultant
Mark A. Richards	Georgia Institute of Technology
Alfred J. Scarpelli	Air Force Research Laboratory

Table A.1: Study Committee Members.

A.2 Biographies

Keren Bergman is a Professor of Electrical Engineering at Columbia University where she also

directs the Lightwave Research Laboratory. Her current research programs involve optical interconnection networks for advanced computing systems, photonic packet switching, and nanophotonic networks-on-chip. Before joining Columbia, Dr. Bergman was with the optical networking group at Tellium where she headed the optical design of large-scale MEMS based cross-connects. Dr. Bergman received her B.S in 1988 from Bucknell University, the M.S. in 1991 and Ph.D. in 1994 from M.I.T. all in Electrical Engineering. She is a recipient of the National Science Foundation CAREER award in 1995 and the Office of Naval Research Young Investigator in 1996. In 1997 she received the CalTech President's Award for joint work with JPL on optical interconnection networks. Dr. Bergman led the optical interconnect effort in the HTMT Petaflops scale design project that combined advanced technologies and parallel architecture exploration. She recently served as Technical Advisor to the Interconnect Thrust of the NSA's Advanced Computing Systems research initiative. Dr. Bergman is a senior member of IEEE and a fellow of OSA. She is currently Associate Editor for IEEE *Photonic Technology Letters* and Editor-in-Chief of the OSA *Journal of Optical Networking*.

Shekhar Borkar graduated with MSc in Physics from University of Bombay, MSEE from University of Notre Dame in 1981, and joined Intel Corporation. He worked on the 8051 family of microcontrollers, the iWarp multicomputer project, and subsequently on Intel's supercomputers. He is an Intel Fellow and director of Microprocessor Research. His research interests are high performance and low power digital circuits, and high-speed signaling.

Daniel P. Campbell is a Senior Research Engineer in the Sensors and Electromagnetic Applications Laboratory of the Georgia Tech Research Institute. Mr. Campbell's research focuses on application development infrastructure for high performance embedded computing, with an emphasis on inexpensive, commodity computing platforms. He is co-chair of the Vector Signal Image Processing Library (VSIPL) Forum, and has developed implementations of the VSIPL and VSIPL++ specifications that exploit various graphics processors for acceleration. Mr. Campbell has been involved in several programs that developed middleware and system abstractions for configurable multicore processors, including DARPA's Polymorphous Computing Architectures (PCA) program.

William W. Carlson is a member of the research staff at the IDA Center for Computing Sciences where, since 1990, his focus has been on applications and system tools for large-scale parallel and distributed computers. He also leads the UPC language effort, a consortium of industry and academic research institutions aiming to produce a unified approach to parallel C programming based on global address space methods. Dr. Carlson graduated from Worcester Polytechnic Institute in 1981 with a BS degree in Electrical Engineering. He then attended Purdue University, receiving the MSEE and Ph.D. degrees in Electrical Engineering in 1983 and 1988, respectively. From 1988 to 1990, Dr. Carlson was an Assistant Professor at the University of Wisconsin-Madison, where his work centered on performance evaluation of advanced computer architectures.

William J. Dally is The Willard R. and Inez Kerr Bell Professor of Engineering and the Chairman of the Department of Computer Science at Stanford University. He is also co-founder, Chairman, and Chief Scientist of Stream Processors, Inc. Dr. Dally and his group have developed system architecture, network architecture, signaling, routing, and synchronization technology that can be found in most large parallel computers today. While at Bell Labs Bill contributed to the BELLMAC32 microprocessor and designed the MARS hardware accelerator. At Caltech he designed the MOSSIM Simulation Engine and the Torus Routing

Chip which pioneered wormhole routing and virtual-channel flow control. While a Professor of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology his group built the J-Machine and the M-Machine, experimental parallel computer systems that pioneered the separation of mechanisms from programming models and demonstrated very low overhead synchronization and communication mechanisms. At Stanford University his group has developed the Imagine processor, which introduced the concepts of stream processing and partitioned register organizations. Dr. Dally has worked with Cray Research and Intel to incorporate many of these innovations in commercial parallel computers, with Avici Systems to incorporate this technology into Internet routers, co-founded Velio Communications to commercialize high-speed signaling technology, and co-founded Stream Processors, Inc. to commercialize stream processor technology. He is a Fellow of the IEEE, a Fellow of the ACM, and a Fellow of the American Academy of Arts and Sciences. He has received numerous honors including the IEEE Seymour Cray Award and the ACM Maurice Wilkes award. He currently leads projects on computer architecture, network architecture, and programming systems. He has published over 200 papers in these areas, holds over 50 issued patents, and is an author of the textbooks, *Digital Systems Engineering and Principles and Practices of Interconnection Networks*.

Monty Denneau is a Research Staff Member at the IBM T.J. Watson Research Center in Yorktown Heights, NY. He has been involved in the architecture, design, and construction of a number of special and general purpose parallel machines.

Paul D. Franzon is currently a Professor of Electrical and Computer Engineering at North Carolina State University. He earned his Ph.D. from the University of Adelaide, Adelaide, Australia in 1988. He has also worked at AT&T Bell Laboratories, DSTO Australia, Australia Telecom and two companies he cofounded, Communica and LightSpin Technologies. His research interests focus on three main areas: Interconnect solutions; Application Specific Architectures and Nanocomputing circuits and structures. He developed core concepts in low-power contactless signaling (capacitive and inductive coupling) for conventional and 3D systems. Examples of these systems have flown in test satellites. He has developed MEMS-based interconnect solutions for electronic and optical applications. He led the development of the popular Spice2Ibis tools and IC Physical Design Kits that have thousands of users world-wide. He has developed sub-25 nm wafer-scale interconnect solutions and new approaches to using nano-crystal elements in electronics. He has established new approaches to enabling extreme environment circuits. He has lead several major research efforts and published over 180 papers in these areas. In 1993 he received an NSF Young Investigators Award, in 2001 was selected to join the NCSU Academy of Outstanding Teachers, in 2003, selected as a Distinguished Alumni Professor, and in 2005 won the Alcoa Research award. He is a Fellow of the IEEE.

William Harrod joined DARPA's Information Processing Technology Office (IPTO) as a Program Manager in December of 2005. His area of interest is extreme computing, including a current focus on advanced computer architectures and system productivity, including self-monitoring and self-healing processing, Exascale computing systems, highly productive development environments and high performance, advanced compilers. He has over 20 years of algorithmic, application, and high performance processing computing experience in industry, academics and government. Prior to his DARPA employment, he was awarded a technical fellowship for the intelligence community while employed at Silicon Graphics Incorporated (SGI). Prior to this at SGI, he led technical teams developing specialized processors and ad-

vanced algorithms, and high performance software. Dr. Harrod holds a B.S. in Mathematics from Emory University, a M.S. and a Ph.D. in Mathematics from the University of Tennessee.

Kerry L. Hill is a Senior Electronics Engineer with the Advanced Sensor Components Branch, Sensors Directorate, Air Force Research Laboratory, Wright-Patterson AFB OH. Ms. Hill has 27 years experience in advanced computing hardware and software technologies. Her current research interests include advanced digital processor architectures, real-time embedded computing, and reconfigurable computing. Ms. Hill worked computer resource acquisition technical management for both the F-117 and F-15 System Program Offices before joining the Air Force Research Laboratory in 1993. Ms. Hill has provided technical support to several DARPA programs including Adaptive Computing Systems, Power Aware Computing and Communications, Polymorphous Computing Architectures, and Architectures for Cognitive Information Processing.

Jon C. Hiller is a Senior Program Manager at Science and Technology Associates, Inc. Mr. Hiller has provided technical support to a number of DARPA programs, and specifically computing architecture research and development. This has included the Polymorphous Computing Architectures, Architectures for Cognitive Information Processing, Power Aware Computing and Communications, Data Intensive Systems, Adaptive Computing Systems, and Embedded High Performance Computing Programs. Previously in support of DARPA and the services, Mr. Hiller's activities included computer architecture, embedded processing application, autonomous vehicle, and weapons systems research and development. Prior to his support of DARPA, Mr. Hiller worked at General Electric's Military Electronic Systems Operation, Electronic Systems Division in the areas of GaAs and CMOS circuit design, computing architecture, and digital and analog design and at Honeywell's Electro-Optics Center in the areas of digital and analog design. Mr. Hiller has a BS from the University of Rochester and a MS from Syracuse University in Electrical Engineering.

Sherman Karp has been a consultant to the Defense Research Projects Agency (DARPA) for the past 21 years and has worked on a variety of projects including the High Productivity Computing System (HPCS) program. Before that he was the Chief Scientist for the Strategic Technology Office (STO) of DARPA. At DARPA he did pioneering work in Low Contrast (sub-visibility) Image enhancement and Multi-Spectral Processing. He also worked in the area of fault tolerant spaceborne processors. Before moving to DARPA, he worked at the Naval Ocean Systems Center where he conceived the concept for Blue-Green laser communications from satellite to submarine through clouds and water, and directed the initial proof of principle experiment and system design. He authored two seminal papers on this topic. For this work he was named the NOSC Scientist of the Year (1976), and was elected to the rank of Fellow in the IEEE. He is currently a Life Fellow. He has co-authored four books and two Special Issues of the IEEE. He was awarded the Secretary of Defense Medal for Meritorious Civilian Service, and is a Fellow of the Washington Academy of Science, where he won the Engineering Sciences Award. He was also a member of the Editorial Board of the IEEE Proceedings, the IEEE FCC Liaison Committee, the DC Area IEEE Fellows Nomination Committee, the IEEE Communications Society Technical Committee on Communication Theory, on which he served as Chairman from 1979-1984, and was a member of the Fellows Nominating Committee. He is also a member of Tau Beta Pi, Eta Kappa Nu, Sigma Xi and the Cosmos Club.

Stephen W. Keckler is an Associate Professor of both Computer Sciences and Electrical and Computer Engineering at the University of Texas at Austin. He earned his Ph.D. from the

Massachusetts Institute of Technology in 1998. Dr. Keckler's research focuses on three main areas: high performance parallel processor architectures, distributed memory architectures, and interconnection networks. At MIT, he worked on the M-Machine, an experimental parallel computer system that pioneered multithreaded and tightly-coupled multicore processors. At UT-Austin, his group developed the NUCA cache which exploits non-uniform access time inherent in wire-dominated integrated circuits. His group recently built the TRIPS machine, a parallel computing system with numerous innovations in the processing and memory subsystems. The TRIPS machine demonstrates a hybrid dataflow instruction set and execution model and employs a logically and physically tiled implementation. Other innovations include tight coupling of routed interconnection networks into processing cores and reconfigurability to allow automatic or manual control of the memory hierarchy. Dr. Keckler has also worked as a VLSI circuit designer at Intel. Dr. Keckler has won numerous awards, including an Alfred P. Sloan Research Fellowship, the ACM Grace Murray Hopper award, an NSF CAREER award, and the 2007 President's Associates Teaching Excellence Award at UT-Austin. Dr. Keckler is a senior member of both the IEEE and the ACM, and a member of Sigma Xi and Phi Beta Kappa.

Dean Klein is the Vice President of Memory System Development at Micron Technology, Inc., where he has held a variety of executive positions since joining Micron in 1999. Mr. Klein's earliest role at Micron was the development of embedded DRAM and logic capability at Micron, a provider of semiconductor memory devices. The embedded DRAM efforts culminated in the creation of the Yukon Processor-In-Memory device, an embedded DRAM part containing 16MBytes of commodity-density DRAM closely coupled to a SIMD array of 256 processing elements. Prior to joining Micron Technology, Klein held the position of Chief Technical Officer and EVP at Micron Electronics, Inc. a personal computer manufacturer. While at Micron Electronics, Klein was responsible for the development of chip sets that exploited advanced memory technology to dramatically boost the performance of Intel's highest performing processors. Prior to Micron Electronics, Mr. Klein was President of PC Tech, Inc., which he co-founded in 1984 and which became a wholly owned subsidiary of Micron Electronics in 1995. Mr. Klein is a graduate of the University of Minnesota, with Bachelor's and Master's degrees in Electrical Engineering. He holds over 180 patents in the area of computer architecture and memory.

Peter M. Kogge (chair) is currently the Associate Dean for research for the College of Engineering, the Ted McCourtney Chair in Computer Science and Engineering, and a Concurrent Professor of Electrical Engineering at the University of Notre Dame, Notre Dame, Indiana. From 1968 until 1994, he was with IBM's Federal Systems Division in Owego, NY, where he was appointed an IBM Fellow in 1993. In 1977 he was a Visiting Professor in the ECE Dept. at the University of Massachusetts, Amherst, MA, and from 1977 through 1994, he was also an Adjunct Professor of Computer Science at the State University of New York at Binghamton. He has been a Distinguished Visiting Scientist at the Center for Integrated Space Microsystems at JPL, and the Research Thrust Leader for Architecture in Notre Dame's Center for Nano Science and Technology. For the 2000-2001 academic year he was also the Interim Schubmehl-Prein Chairman of the CSE Dept. at Notre Dame. His research areas include advanced VLSI and nano technologies, non von Neumann models of programming and execution, parallel algorithms and applications, and their impact on massively parallel computer architecture. Since the late 1980s' this has focused on scalable single VLSI chip designs integrating both dense memory and logic into "Processing In Memory" (PIM) archi-

tures, efficient execution models to support them, and scaling multiple chips to complete systems, for a range of real system applications, from highly scalable deep space exploration to trans-petaflops level supercomputing. This has included the world's first true multi-core chip, EXECUBE, that in the early 1990s integrated 4 Mbits of DRAM with over 100K gates of logic to support a complete 8 way binary hypercube parallel processor which could run in both SIMD and MIMD modes. Prior parallel machines included the IBM 3838 Array Processor which for a time was the fastest single precision floating point processor marketed by IBM, and the Space Shuttle Input/Output Processor which probably represents the first true parallel processor to fly in space, and one of the earliest examples of multi-threaded architectures. His Ph.D. thesis on the parallel solution of recurrence equations was one of the early works on what is now called parallel prefix, and applications of those results are still acknowledged as defining the fastest possible implementations of circuits such as adders with limited fan-in blocks (known as the Kogge-Stone adder). More recent work is investigating how PIM-like ideas may port into quantum cellular array (QCA) and other nanotechnology logic, where instead of "Processing-In-Memory" we have opportunities for "Processing-In-Wire" and similar paradigm shifts.

Robert F. Lucas is the Director of the Computational Sciences Division of the University of Southern California's Information Sciences Institute (ISI). There he manages research in computer architecture, VLSI, compilers and other software tools. Prior to joining ISI, he was the Head of the High Performance Computing Research Department in the National Energy Research Scientific Computing Center (NERSC) at Lawrence Berkeley National Laboratory. There he oversaw work in scientific data management, visualization, numerical algorithms, and scientific applications. Prior to joining NERSC, Dr. Lucas was the Deputy Director of DARPA's Information Technology Office. He also served as DARPA's Program Manager for Scalable Computing Systems and Data-Intensive Computing. From 1988 to 1998 he was a member of the research staff of the Institute for Defense Analysis, Center for Computing Sciences. From 1979 to 1984 he was a member of the Technical Staff of the Hughes Aircraft Company. Dr. Lucas received his BS, MS, and PhD degrees in Electrical Engineering from Stanford University in 1980, 1983, and 1988 respectively.

Mark A. Richards is a Principal Research Engineer and Adjunct Professor in the School of Electrical and Computer Engineering, Georgia Institute of Technology. From 1988 to 2001, Dr. Richards held a series of technical management positions at the Georgia Tech Research Institute, culminating as Chief of the Radar Systems Division of GTRI's Sensors and Electromagnetic Applications Laboratory. From 1993 to 1995, he served as a Program Manager for the Defense Advanced Research Projects Agency's (DARPA) Rapid Prototyping of Application Specific Signal Processors (RASSP) program, which developed new computer-aided design (CAD) tools, processor architectures, and design and manufacturing methodologies for embedded signal processors. Since the mid-1990s, he has been involved in a series of programs in high performance embedded computing, including the efforts to develop the Vector, Signal, and Image Processing Library (VSIPL) and VSIPL++ specifications and the Stream Virtual Machine (SVM) middleware developed under DARPA's Polymorphous Computing Architectures (PCA) program. Dr. Richards is the author of the text *Fundamentals of Radar Signal Processing* (McGraw-Hill, 2005).

Alfred J. Scarpelli is a Senior Electronics Engineer with the Advanced Sensor Components Branch, Sensors Directorate, Air Force Research Laboratory, Wright-Patterson AFB OH. His current research areas include advanced digital processor architectures, real-time embedded

computing, and reconfigurable computing. Mr. Scarpelli has 32 years research experience in computer architectures and computer software. In the 1970's, he conducted benchmarking to support development of the MIL-STD-1750 instruction set architecture, and test and evaluation work for the DoD standard Ada language development. In the 1980's, he was involved with the DoD VHSIC program, and advanced digital signal processor development, a precursor to the F-22 Common Integrated Processor. In the 1990's, his research focused on the DARPA Pilot's Associate, the development of an embedded, artificial intelligence processor powered by an Associative Memory CoProcessor, real-time embedded software schedulability techniques, VHDL compilation and simulation tools, and application partitioning tools for reconfigurable computing platforms. Since 1997, he has provided technical support to multiple DARPA programs such as Adaptive Computing Systems, Polymorphous Computing Architectures, Architectures for Cognitive Information Processing, Networked Embedded Systems Technology, Mission Specific Processing, and Foliage Penetration. He holds a B.S. degree in Computer Science from the University of Dayton (1979) and an M.S. degree in Computer Engineering from Wright State University (1987).

Steve Scott is the Chief Technology Officer and SVP at Cray Inc., where he has been since 1992 (originally with Cray Research and SGI). Dr. Scott was one of the architects of the groundbreaking Cray T3E multiprocessor, focusing on the interconnect and on synchronization and communication mechanisms. He was the chief architect of the GigaRing system area network used in all Cray systems in the late 1990s. More recently, he was the chief architect of the Cray X1/X1E supercomputers, which combined high performance vector processors with a scalable, globally-addressable system architecture. He was also the chief architect of the next generation Cray "BlackWidow" system, and the architect of the router used in Cray XT3 MPP and the follow-on Baker system. Dr. Scott is currently leading the Cray Cascade project, which is part of the DARPA High Productivity Computing Systems program targeting productive, trans-petaflop systems in the 2010 timeframe. Dr. Scott received his PhD in computer architecture from the University of Wisconsin, Madison in 1992, where he was a Wisconsin Alumni Research Foundation and Hertz Foundation Fellow. He holds seventeen US patents, has served on numerous program committees, and has served as an associate editor for the IEEE Transactions on Parallel and Distributed Systems. He was the recipient of the ACM 2005 Maurice Wilkes Award and the IEEE 2005 Seymour Cray Computer Engineering Award.

Allan E. Snively is an Adjunct Assistant Professor in the University of California at San Diego's Department of Computer Science and is founding director of the Performance Modeling and Characterization (PMaC) Laboratory at the San Diego Supercomputer Center. He is a noted expert in high performance computing (HPC). He has published more than 50 papers on this subject, has presented numerous invited talks including briefing U.S. congressional staff on the importance of the field to economic competitiveness, was a finalist for the Gordon Bell Prize 2007 in recognition for outstanding achievement in HPC applications, and is primary investigator (PI) on several federal research grants. Notably, he is PI of the Cyberinfrastructure Evaluation Center supported by National Science Foundation, and Co-PI in charge of the performance modeling thrust for PERI (the Performance Evaluation Research Institute), a Department of Energy SciDAC2 institute.

Thomas Sterling is the Arnaud and Edwards Professor of Computer Science at Louisiana State University and a member of the Faculty of the Center for Computation and Technology. Dr. Sterling is also a Faculty Associate at the Center for Computation and Technology at Cali-

fornia Institute of Technology and a Distinguished Visiting Scientist at Oak Ridge National Laboratory. Sterling is an expert in the field of parallel computer system architecture and parallel programming methods. Dr. Sterling led the Beowulf Project that performed seminal pathfinding research establishing commodity cluster computing as a viable high performance computing approach. He led the Federally sponsored HTMT project that conducted the first Petaflops scale design point study that combined advanced technologies and parallel architecture exploration as part of the national petaflops initiative. His current research directions are the ParalleX execution model and processor in memory architecture for directed graph based applications. He is a winner of the Gordon Bell Prize, co-author of five books, and holds six patents.

R. Stanley Williams is an HP Senior Fellow at Hewlett-Packard Laboratories and Director of the Advanced Studies Lab (ASL), a group of approximately 80 research scientists and engineers working in areas of strategic interest to HP. The areas covered in ASL include computing architectures, photonics, nano-electronics, micro- and nano-mechanical systems, information theory and quantum information processing. He received a B.A. degree in Chemical Physics in 1974 from Rice University and his Ph.D. in Physical Chemistry from U. C. Berkeley in 1978. He was a Member of Technical Staff at AT&T Bell Labs from 1978-80 and a faculty member (Assistant, Associate and Full Professor) of the Chemistry Department at UCLA from 1980-1995. His primary scientific research during the past thirty years has been in the areas of solid-state chemistry and physics, and their applications to technology. Most recently, he has examined the fundamental limits of information and computing, which has led to his current research in nano-electronics and nano-photonics. He has received awards for business, scientific and academic achievement, including the 2004 Joel Birnbaum Prize (the highest internal HP award for research), the 2004 Herman Bloch Medal for Industrial Research, the 2000 Julius Springer Award for Applied Physics, the 2000 Feynman Prize in Nanotechnology. He was named to the inaugural Scientific American 50 Top Technology leaders in 2002 and then again in 2005 (the first to be named twice). In 2005, the US patent collection that he has assembled at HP was named the world's top nanotechnology intellectual property portfolio by Small Times magazine. He was a co-organizer and co-editor (with Paul Alivisatos and Mike Roco) of the workshop and book "Vision for Nanotechnology in the 21st Century", respectively, that led to the establishment of the U. S. National Nanotechnology Initiative in 2000. He has been awarded more than 60 US patents with more than 40 pending and he has published over 300 papers in reviewed scientific journals. One of his patents on crossbar based nanoscale circuits was named as one of five that will "transform business and technology" by MIT's Technology Review in 2000.

Katherine A. Yelick is Professor of Electrical Engineering and Computer Sciences at the University of California, Berkeley and a Senior Research Scientist at the Lawrence Berkeley National Laboratory. Prof. Yelick co-leads and co-invented the Titanium language, which is a Partitioned Global Address Space (PGAS) language based on Java. The Titanium group has demonstrated tremendous productivity advantages for adaptive mesh refinement algorithms and immersed boundary method simulations. She also leads the Berkeley UPC project, an open source compiler project for the Unified Parallel C (UPC) language. She co-invented the UPC language with 5 other researchers from IDA, LLNL, and UC Berkeley, and co-authored both the original language specification and the main UPC textbook, *UPC: Distributed Shared-Memory Programming* (Wiley-Interscience, 2005). The Berkeley UPC compiler project is a highly portable compiler that is used on clusters and shared memory

systems and is shipped with systems from Cray, SGI, and some Linux clusters. Prof. Yelick leads the Berkeley Institute for Performance Studies (BIPS), which involves performance analysis, modeling, tuning, and benchmarking. The groups within BIPS work on large application performance studies across vector, cluster, and ultrascale (BG/L) supercomputers as well as synthetic benchmarking and identification of architectural bottlenecks. She also co-leads the Berkeley Benchmarking and Optimization (BeBOP) group, which developed the OSKI system for automatically tuning sparse matrix kernels. Based on ideas from her earlier Sparsity system, OSKI includes optimizations for registers and caches that are tailored to a given sparse matrix structure. The group has recently developed multicore optimizations which are being integrated into OSKI. Prof. Yelick received her B.S., M.S., and Ph.D. degrees in Computer Science from the Massachusetts Institute of Technology. She has been a Visiting Researcher at ETH, Zurich and a Visiting Associate Professor at MIT, and has received teaching awards from both Berkeley and from MIT. She was a member of the WTEC team on "Assessment of High-End Computing Research and Development in Japan," and is currently a member of an NRC committee on "Sustaining the Growth in Computing Performance."

This page intentionally left blank.

Appendix B

Exascale Computing Study Meetings, Speakers, and Guests

B.1 Meeting #1: Study Kickoff

May 30, 2007, Arlington, VA

Host: Science and Technology Associates

Committee members present: Shekhar Borkar, Dan Campbell, William Dally, Monty Denneau, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Steve Keckler, Dean Klein, Peter Kogge, Bob Lucas, Mark Richards, Alfred Scarpelli, Steve Scott, Allan Snavey, Thomas Sterling, Kathy Yelick

Visitors

- Jose Munoz, NSF
- Rob Schreiber, HP Labs
- Barbara Yoon, DARPA

Presentations

- Peter Kogge - Introduction and Goals
- Thomas Sterling - Towards an Exascale Report
- William Dally - NRC Future of Supercomputing Study
- Dean Klein - Memory
- Steve Scott - Processor Requirements and Scaling
- Allan Snavey - Application Scaling
- Katherine Yelick - Berkeley Dwarfs

B.2 Meeting #2: Roadmaps and Nanotechnology

June 26-27, 2007, Palo Alto, CA

Host: Hewlett-Packard Laboratories

Committee members present: Shekhar Borkar, Daniel Campbell, William Carlson, William Dally, Monty Denneau, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Peter Kogge, Mark Richards, Allan Snaveley, Stanley Williams, Katherine Yelick

Visitors

- Jeff Draper, USC/ISI
- Rob Smith, Nantero
- Norm Jouppi, HP Labs
- Richard Kaufmann, HP Labs
- Phil Kuekes, HP Labs
- Chandrakant Patel, HP Labs
- Rob Schreiber, HP Labs
- Greg Snider, HP Labs

Presentations

- Shekhar Borkar - Logic Roadmap
- Norm Jouppi - Configurable Isolation
- Stan Williams: Exascale Overview
- Greg Snider - Adaptive, Probabilistic Exacomputing
- William Dally - Future Projections Spreadsheet
- Phil Kuekes - Defects & Faults
- Richard Kaufmann - Checkpoint/Restart & Ratios
- Steve Keckler - Reliability
- Chandrakant Patel - Smart Data Center

B.3 Special Topics Meeting #1: Packaging

July 17-18, 2007, Atlanta, GA

Host: Georgia Institute of Technology

Committee members present: William Dally, Dan Campbell, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Peter Kogge, Mark Richards, Alfred Scarpelli. By teleconference: Kerry Hill, Sherman Karp

Visitors

- Muhannad Bakir, Georgia Institute of Technology
- Robert Conn, Research Triangle Institute
- Patrick Fay, University of Notre Dame
- Paul Franzon, NCSU
- Dorota Temple, Research Triangle Institute
- Rao Tummala, Georgia Institute of Technology

Presentations

- Paul Franzon - High Bandwidth Interconnect
- Rao Tummala - System Packaging
- Robert Conn - 3D Impact on HPC
- Dorota Temple - 3D Integration
- Patrick Fay - Quilt Packaging
- Muhannad Bakir - Electrical, Optical & Thermofluidic Interconnects

B.4 Meeting #3: Logic

July 24-25, 2007, Portland, OR

Host: Intel Corporation

Committee members present: Shekhar Borkar, Daniel Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Peter Kogge, Robert Lucas, Mark Richards, Steve Scott, Allan Snavely, Stanley Williams

Visitors

- Jim Held, Intel
- Jose Maiz, Intel
- Tim Mattson, Intel
- Marko Radosavljevic, Intel

Presentations

- William Dally - Exascale Silicon Architecture
- Allan Snavely - Applications
- Shekhar Borkar - Exascale Power Performance Tradeoffs
- Steve Scott - Socket Architecture
- Stanley Williams - Nanoscale Implications for Exascale

- Steve Keckler - Reliability
- Paul Franzon - 3-D Interconnects
- Dean Klein - DRAM Challenges
- Marko Radosavljevic - Nanoelectronic Devices
- Jim Held - Terascale Computing
- Jose Maiz - Exascale Reliability
- Clair Webb - 3DIC Integration
- Tim Mattson - Programming at Exascale

B.5 Meeting #4: Memory Roadmap and Issues

August 16-17, Boise, ID

Host: Micron Technology

Committee members present: Shekhar Borkar, Daniel Campbell, Monty Denneau, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Peter Kogge, Robert Lucas, Mark Richards, Steve Scott, Allan Snavey, Stanley Williams. By teleconference: Paul Franzon

Visitors

- Rob Schreiber, HP Labs
- Jim Hutchby, Semiconductor Research Corporation
- Terry Lee, Micron
- Dave Resnick, Micron
- Kevin Ryan, Micron
- Brent Keeth, Micron
- Mark Durcan, Micron
- Kirk Prall, Micron
- Chandra Mouli, Micron

Presentations

- Paul Franzon - 3D Memory Packaging
- Jim Hutchby - Emerging Research Memory Devices
- Steve Scott - Thoughts on a 3D Node
- Allan Snavey - Locality versus performance
- Monty Denneau - EDRAM
- Steve Scott - Iso Curves

- Dean Klein - Micron's Yukon Architecture
- Kirk Prall - NAND
- Overview Brent Keeth - DRAM Architectures & Technology
- Micron - 3D Integration Update
- Chandra Mouli - Memory Technology Trends
- Micron - Low End Memory Solutions

B.6 Special Topics Meeting #2: Architectures and Programming Environments

August 29-30, 2007, Palo Alto, CA

Host: Stanford University

Committee members present: Shekhar Borkar, Daniel Campbell, William Dally, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Peter Kogge, Robert Lucas, Mark Richards, Alfred Scarpelli, Steve Scott, Allan Snavey, Thomas Sterling, Stanley Williams, Katherine Yelick

Visitors

- Krste Asanovic, University of California at Berkeley
- Luiz Barroso, Google
- Mark Horowitz, Stanford University
- Kunle Olukotun, Stanford University
- Mary Hall, University of Southern California
- Vivek Sarkar, Rice University

Presentations

- Allan Snavey - Isosurfaces
- Stephen Keckler - Reliability for Exascale
- Robert Lucas - Musings on Exascale
- Robert Lucas - ORNL Exascale presentation
- Katherine Yelick - Memory footprint
- William Dally - Strawman Architecture
- Steve Scott - 3D Node Thoughts
- Stephen Keckler - IBM Fault Tolerance from HotChips
- Mark Horowitz - Power & CMOS Scaling

B.7 Special Topics Meeting #3: Applications, Storage, and I/O

September 6-7, 2007, Berkeley, CA

Host: University of California at Berkeley

Committee members present: Daniel Campbell, William Dally, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Dean Klein, Peter Kogge, Robert Lucas, Mark Richards, Alfred Scarpelli, Allan Snaveley, Thomas Sterling

Visitors

- Dave Koester, MITRE
- Winfried Wilcke, IBM
- Garth Gibson, Carnegie Mellon University
- Dave Aune, Seagate Technology
- Gary Grider, Los Alamos National Laboratory
- Duncan Stewart, HP Labs
- Dave Bailey, Lawrence Berkeley Laboratory
- John Shalf, Lawrence Berkeley Laboratory
- Steve Miller, NetApp

Presentations

- David Koester - Application Footprints
- Garth Gibson - Petascale Failure Data
- Gary Grider - Gaps
- Dave Aune - Storage Trends
- Dean Klein - Memory Resiliency
- Gary Grider - ASC I/O Report
- John Shalf - I/O Requirements
- Duncan Stewart - Nano-crosspoint Memory

B.8 Special Topics Meeting #4: Optical Interconnects

September 25-26, 2007, Palo Alto, CA

Host: Stanford University

Committee members present: Daniel Campbell, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Peter Kogge, Robert Lucas, Steve Scott, Mark Richards, Alfred Scarpelli, Allan Snaveley, Thomas Sterling

Visitors

- Ravi Athale, MITRE
- Karen Bergman, Columbia University
- Alex Dickinson, Luxtera
- David Miller, Stanford University
- Dave Koester, MITRE
- Bill Wilson, InPhase
- Mark Beals, Massachusetts Institute of Technology
- Cheng Hengju, Intel
- Krishna Saraswat, Stanford University
- Alan Benner, IBM
- Jeff Kash, IBM
- Ahok Krishnamoorthy, Sun
- Ray Beausoleil, HP Labs
- Mootaz Elnohazy, IBM

Presentations

- David Koester - Application Scaling Requirements
- Stephen Keckler - Reliability
- Mark Beals - Photonic Integration
- Jeffrey Kash & Alan Benner - Optical/electrical Interconnect Technologies
- Bill Wilson - Holographic Archive
- Ray Beausoleil - Nanophotonic Interconnect
- Krishna Saraswat - Optics and CNT
- Keren Bergman - Photonic Interconnects

B.9 Meeting #5: Report Conclusions and Finalization Plans

October 10-11, 2007, Marina del Rey, CA

Host: University of Southern California Information Sciences Institute

Committee members present: Shekhar Borkar, Daniel Campbell, William Carlson, William Dally, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Peter Kogge, Robert Lucas, Mark Richards, Alfred Scarpelli, Allan Snively, Katherine Yelick

Visitors

- Keren Bergman, Columbia University
- Loring Craymer, University of Southern California Information Sciences Institute
- Phil Kuekes, HP Labs

This page intentionally left blank.

Appendix C

Glossary and Abbreviations

AMB: Advanced Memory Buffer

AMR: Adaptive Mesh Refinement

AVUS: Air Vehicle Unstructured Solver

BCH: , Bose, Chaudhuri, Hocquenghem error correcting code.

BER: Bit Error Rate

BIST: Built-In-Self-Test

bitline: The bitline receives or delivers charge from the memory cell capacitor through the memory cell FET access device. This charge is sensed and driven by the sense amplifier.

BGA: Ball Grid Array

BER: Bit Error Rate

CAD: Computer-aided Design

CAGR: Compound Annual Growth Rate

CDR: Clock and Data Recovery
Computational Fluid Dynamics

CMOL: CMOS MOlecular Logic

CMOS: a common type of logic that employs two types of transistors whose on/off characteristics are essentially opposite.

CMP: Chip Multi-Processor

CNT: Carbon Nano Tubes

CSP: Communicating Sequential Process model

DDR: Double Data Rate DRAM. A protocol for memory chips that has a higher bit signalling rate than earlier types.

Digitline see bitline.

DIMM: Dual Inline Memory Module. The common form of packaging for memory devices. DIMM's are available for all commodity main memory types, with and without ECC for applications from desktop computers to supercomputers.

DRAM: Dynamic Random Access Memory. A memory typically composed of a one transistor, one capacitor memory cell. This memory is most commonly used as main memory in a computing system.

DWDM: Dense Wavelength Division Multiplexing

DSP: Digital Signal Processor

E3SGS: Simulation and Modeling at exascale for Energy, Ecological Sustainability and Global Security

EB: exabyte

ECC: Error Correcting Code

eDRAM: embedded Dynamic Random Access Memory

EIP: Exa Instructions Processed to completion

EIPs: Exa Instructions Processed per second

E/O: Electrical to Optical

EOS/DIS: Earth Observing System/Data Information System

FB-DIMM: Fully Buffered Dual Inline Memory Module

Fe-RAM: Ferro-electric RAM

FG: Floating Gate.

FINFET: a dual gate non-planar transistor where the major structures look like "fins" on top of the substrate.

FIT: Failures in Time. The number of expected failures in a device over a billion hour of operation. Testing for this is generally done by accelerated testing a million devices for a thousand hours. Typically performed on memory devices.

flop: floating point operation

FPGA: Field Programmable Gate Array

FPNI: Field-Programmable Nanowire Interconnect

FPU: Floating Point Unit

GAS: Global Address Space

GB: giga byte

GUPS: Global Updates Per Second

HAL: Hardware Abstraction Layer

HECRTF: High End Computing Revitalization Task Force

HT: Hyper Transport

HPC: High Performance Computing

HPCS: High Productivity Computing Systems - a DARPA program

HPL: High Performance Linpack benchmark

HTMT: Hybrid Technology Multi-Threaded architecture

HVAC: Heating, Ventilating, and Air Conditioning

ILP: Instruction Level Parallelism

IMT: Interleaved Multi-Threading

IPS: Instructions Per Second

ISA: Instruction Set Architecture

JEDEC: Joint Electron Device Engineering Council. A standards committee for many commercial commodity electronic parts.

JJ Josephson Junction

MEMS: Micro Electrical Mechanical Systems

MLC: Multi-Level-Cell. A technology for storing more than one bit in a NAND Flash cell.

MOS: see MOSFET

MOSFET: Metal-Oxide-Semiconductor Field Effect Transistor. While not totally accurate (gates today are not implemented from metal), this is common name for the typical transistor used today.

MPI: Message Passing Interface

MPP: Massively Parallel Processor

MRAM: Magnetic Random Access Memory

MTJ: magnetic tunnel junctions

MTTI: Mean Time To Interrupt

MW: Mega Watt

NAS: National Academy of Science

nm: nano meter

NoC: Network on Chip

NVRAM: Non Volatile RAM

O/E: Optical to Electrical

PB: Peta Byte

PCB: Printed Circuit Board

PCRAM: Phase-Change RAM

PDE: Partial Differential Equation

PDU: Power Distribution Unit

pGAS: Partitioned global address space

PIM: Processing-In-Memory

pJ: pico joules, or 10^{-12} joules

PSU: Power Supply Unit

QK: Quintessential Kernel

qubit: quantum bit

RAM: Random Access Memory

RLDRAM: Reduced Latency DRAM

RMA: Reliability, Maintainability, and Availability

ROM: Read Only Memory

RRAM: Resistive RAM

RSFQ: Rapid Single Flux Quantum Device

RTL: Register Transfer Language

SCI: System Call Interface

SCM: System Capable Memory

SCP: Single Chip Package

SDRAM: Synchronous DRAM

SECCDED: Single Error Correcting Double Error Detecting code

SEM: Scanning Electron Microscope

SER: Soft Error Rate. Failures in a component or system that are transient failures can be aggregated to compute a soft error rate. SER sources include noise and ionizing radiation.

SerDes: Serializer/Deserializer

SEU: Single Event Upset

SIMD: Single Instruction, Multiple Data

SLC: Single-Level-Cell. A flash with one bit of information stored per cell.

SMT: Simultaneous Multi-Threading

SNIC: Semiconductor Nanowire InterConnect

SNR: Signal to Noise Ratio

SOC: System On a Chip

SOI: Silicon On Insulator

SONOS: Semiconductor-Oxide-Nitride-Oxide-Semiconductor memory

SPMD: Single Program Multiple Data

SRAM: static random access memory. A memory typically composed of a six transistor storage cell. These cells can be fast and are typically used in processor caches where access time is most critical.

SSTL: Stub Series Terminated Logic signalling standard

TEC: Thermal Electric Coolers

TIA: Trans-Impedance Amplifier - a type of optical receiver

TLB: translation lookaside buffer

TLC: Thread Level Parallelism

TMR: Triple Modular Redundancy

TSV: Through Silicon Via

UPS: Uninterruptible Power Supply

VFS: Virtual File System interface

VR: Voltage Regulator

VRT: Variable Retention Time

V_{dd} : Voltage drain to drain - main operation voltage for silicon circuits

VSEL: Vertical-Cavity Surface-Emitting Laser

Wordline: The signal line that drives the gates of the memory cell FET. The wordline may be divided across several sub-arrays, but must be considered as the logical sum of its parts. Typically a wordline activates a row of 8K memory cells.

WRF: Weather Research and Forecasting code.

This page intentionally left blank.

Bibliography

- [1] BGA-scale stacks comprised of layers containing integrated circuit die and a method for making the same. US Patent Number 20070158805.
- [2] Historical Notes about the Cost of Hard Drive Storage Space. <http://www.littletechshoppe.com/ns1625/winchest.html>.
- [3] Superconducting Technology Assessment. Technical report, National Security Agency Office of Corporate Assessments, August 2005.
- [4] High Productivity Computer Systems. <http://www.highproductivity.org/>, 2007.
- [5] N.R. Adiga and et al. An Overview of the BlueGene/L Supercomputer. In *ACM/IEEE Conference on Supercomputing*, November 2002.
- [6] E. Adler and et al. The evolution of IBM CMOS DRAM technology. *IBM J. Research and Development*, 39(1/2):pp. 167–188, January 1995.
- [7] Guy AlLee, Milan Milenkovic, and James Song. Data Center Energy Efficiency. http://download.intel.com/pressroom/kits/research/poster_Data_Center_Energy_Efficiency.pdf, June 2007.
- [8] George Almási, Călin Caşcaval, nos José G. Casta Monty Denneau, Derek Lieber, José E. Moreira, and Henry S. Warren. Dissecting Cyclops: a detailed analysis of a multithreaded architecture. *SIGARCH Comput. Archit. News*, 31(1):26–38, 2003.
- [9] C. J. Amsinck, N. H. Di Spigna, D. P. Nackashi, , and P. D. Franzon. Scaling constraints in nanoelectronic random-access memories. *Nanotechnology*, 16, October.
- [10] Ken Anderson, Edeline Fotheringham, Adrian Hill, Bradley Sissom, and Kevin Curtis. High speed holographic data storage at 500 Gbit/in.². <http://www.inphase-technologies.com/downloads/pdf/technology/HighSpeedHDS500Gbin2.pdf>.
- [11] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf>, December 2006.
- [12] Computing Research Association. Grand Research Challenges in Information Systems. <http://www.cra.org/reports/gc.systems.pdf>, 2003.
- [13] Semiconductor Industries Association. *International Technology Roadmap for Semiconductors*. 2006.

- [14] I. G. Baek and et al. Multi-layer cross-point binary oxide resistive memory (OxRRAM) for post-NAND storage application. *IEDM*, 2006.
- [15] Gary H. Bernstein, Qing Liu, Minjun Yan, Zhuowen Sun, David Kopp, Wolfgang Porod, Greg Snider, and Patrick Fay. Quilt Packaging: High-Density, High-Speed Interchip Communications. *IEEE Trans. on Advanced Packaging*, 2007.
- [16] B. Black and et al. Die stacking (3d) microarchitecture. volume 39. *IEEE Micro*, 2006.
- [17] Defense Science Board. Task Force on DoD Supercomputer Needs. <http://stinet.dtic.mil/cgi-bin/GetTRDoc?AD=ADA383826&Location=U2&doc=GetTRDoc.pdf>, October 2000.
- [18] Defense Science Board. Report on Joint U.S. Defense Science Board and UK Defence Scientific Advisory Council Task Force on Defense Critical Technologies. http://www.acq.osd.mil/dsb/reports/2006-03-Defense_Critical_Technologies.pdf, March 2006.
- [19] Shekhar Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.
- [20] Arthur A. Bright, Matthew R. Ellavsky², Alan Gara¹, Ruud A. Haring, Gerard V. Kopsay, Robert F. Lembach, James A. Marcella, Martin Ohmacht¹, and Valentina Salapura¹. Creating the BlueGene/L Supercomputer from Low-Power SoC ASICs. pages 188–189, San Francisco, CA, 2005. *ISSCC*.
- [21] R. Brightwell, K. Pedretti, and K.D. Underwood. Initial performance evaluation of the Cray SeaStar interconnect. *13th Symposium on High Performance Interconnects*, pages 51–57, 17-19 Aug. 2005.
- [22] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *ACM SIGGRAPH*, pages 777–786, August 2004.
- [23] P. Bunyk. RSFQ Random Logic Gate Density Scaling for the Next-Generation Josephson Junction Technology. *IEEE Transactions on Applied Superconductivity*, 13(2):496–497, June 2003.
- [24] P. Bunyk, M. Leung, J. Spargo, and M. Dorojevets. Flux-1 RSFQ microprocessor: physical design and test results. *IEEE Transactions on Applied Superconductivity*, 13(2):433–436, June 2003.
- [25] J. Carlstrom and et al. A 40 Gb/s Network Processor with PISC Dataflow Architecture. In *Int. Solid-State Circuits Conf.*, pages 60–67, San Francisco, USA, Feb. 2004.
- [26] L. Carrington, X. Gao, A. Snively, , and R. Campbell. Profile of AVUS Based on Sampled Memory Tracing of Basic Blocks. Users Group Conference on 2005 Users Group Conference, jun.,.
- [27] Tien-Hsin Chao. Holographic Data Storage. <http://www.thic.org/pdf/Jan01/NASAJPL.t-schao.010116.pdf>.
- [28] Y. Chen, G. Y. Jung, D. A. A. Ohlberg, X. M. Li, D. R. Stewart, J. O. Jeppesen, K. A. Nielsen, J. F. Stoddart, and R. S. Williams. Nanoscale molecular-switch crossbar circuits. *Nanotechnology*, 14:462–468, April 2003.

- [29] Y. C. Chen and et al. An Access-transistor-free (OT/1R) non-volatile resistance random access memory (RRAM) using a novel threshold switching, self-rectifying chalcogenide device. *IEDM Report 37.4.1*, 2003.
- [30] H. Cho, P. Kapur, and K.C. Saraswat. Performance Comparison Between Vertical-Cavity Surface-Emitting Laser and Quantum-Well Modulator for Short-Distance Optical Links. *IEEE Photonics Technology Letters*, 18(3):520–522, February 2006.
- [31] UPC Consortium. UPC language specifications v1.2. Technical report, Lawrence Berkeley National Lab, 2005.
- [32] Livermore Software Technology Corporation. Getting Started with LS-DYNA. <http://www.feainformation.com/m-pdf/IntroDyna.pdf>, 2002.
- [33] A. DeHo, S. C. Goldstein, P. J. Kuekes, and P. Lincoln. Nonphotolithographic nanoscale memory density prospects. *IEEE Transactions on Nanotechnology*, 4:215–228, March 2005.
- [34] A. Dehon. Array-based architecture for fet-based, nanoscale electronics. *IEEE Trans. Nanotechnol.*, 2(1):23–32, 2003.
- [35] A. DeHon. Design of programmable interconnect for sublithographic programmable logic array. pages 127–137, Monterey, CA, February 2005. FPGA.
- [36] A. DeHon. Nanowire-based programmable architecture. *ACM J. Emerg. Technol. Comput. Syst.*, 1(2):109–162, 2005.
- [37] A. DeHon and K. K. Likharev. Hybrid cmos/nanoelectronic digital circuits: Devices, architectures, and design automation. pages 375–382, San Jose, CA, November 2005. ICCAD.
- [38] Dell. Data Center Efficiency in the Scalable Enterprise. <http://www.dell.com/downloads/global/power/ps1q07-20070210-CoverStory.pdf>, February 2007.
- [39] Lisa Dhar, Arturo Hale, Howard E. Katz, Marcia L. Schilling, Melinda G. Schnoes, and Fred C. Schilling. Recording media that exhibit high dynamic range for digital holographic data storage. *Optics Letters*, 24(7):487–489, 1999.
- [40] DoD. White Paper DoD Research and Development Agenda For High Productivity Computing Systems. http://www.nitrd.gov/subcommittee/hec/hecrtf-outreach/bibliography/20010611_high_productivity_computing.s.pdf, June 2001.
- [41] DoD. Report on High Performance Computing for the National Security Community. http://www.nitrd.gov/subcommittee/hec/hecrtf-outreach/bibliography/200302_hec.pdf, July 2002.
- [42] Mattan Erez. *Merrimac – High-Performance, Highly-Efficient Scientific Computing with Streams*. PhD thesis, Stanford University, Stanford, California, November 2005.
- [43] Mattan Erez, Nuwan Jayasena, Timothy J. Knight, and William J. Dally. Fault Tolerance Techniques for the Merrimac Streaming Supercomputer. In *SC05*, Seattle, Washington, USA, November 2005.

- [44] John Feo, David Harper, Simon Kahan, and Petr Konecny. ELDORADO. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 28–34, New York, NY, USA, 2005. ACM.
- [45] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Pub., 1999.
- [46] Richard Games. Survey and Analysis of the National Security High Performance Computing Architectural Requirements. http://www.nitrd.gov/subcommittee/hec/hecrtf-outreach/bibliography/20010604_hpc_arch_survey_analysis.f.pdf, June 2001.
- [47] G. Gao, K. Likharev, P. Messina, and T. Sterling. Hybrid Technology Multithreaded Architecture. In *6th Symp. on Frontiers of Massively Parallel Computation*, pages 98–105, 1996.
- [48] A. Gara and et al. Overview of the Bluegene/L system architecture. *IBM J. of Research and Development*, (2/3):195–212, 2005.
- [49] A. Gayasen, N. Vijaykrishnan, and M. J. Irwin. Exploring technology alternatives for nano-scale fpga interconnects. pages 921–926, Anaheim, CA, June 2005. DAC.
- [50] Garth Gibson. Reflections on Failure in Post-Terascale Parallel Computing. In *Int. Conf. on Parallel Processing*.
- [51] N. E. Gilbert and M. N. Kozicki. An embeddable multilevel-cell solid electrolyte memory array. *EEE Journal of Solid-State Circuits*, 42:1383–1391, June 2007.
- [52] S. C. Goldstein and M. Budiu. Nanofabrics: Spatial computing using molecular electronics. pages 178–189, Goteborg, Sweden, 2001. ISCA.
- [53] S. C. Goldstein and D. Rosewater. Digital logic using molecular electronics. page 12.5, San Francisco, CA, February 2002. ISSCC.
- [54] S. Graham and et al. *Getting Up to Speed: The Future of Supercomputing*. National Academies Press, 2004.
- [55] Jim Gray. Building PetaByte Servers. [http://research.microsoft.com/~gray/talks/ Building%20Petabyte%20Databases%20\(CMU\).ppt](http://research.microsoft.com/~gray/talks/Building%20Petabyte%20Databases%20(CMU).ppt).
- [56] Y.J. E. Green, J. W. Choi, A. Boukai, Y. Bunimovich, E. Johnston-Halperin, E. DeIonno, Y. Luo, B. A. Sheriff, K. Xu, Y. S. Shin, H. R. Tseng, J. F. Stoddart, and J. R. Heath. A 160-kilobit molecular electronic memory patterned at 10^{11} bits per square centimetre. *Nature*, 445:414–417, January 25 2005.
- [57] Michael Gschwind. Chip multiprocessing and the cell broadband engine. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 1–8, New York, NY, USA, 2006. ACM.
- [58] D. Guckenberger, J.D. Schaub, D. Kucharski, and K.T. Komegay. 1 V, 10 mW 10 Gb/s CMOS Optical Receiver Front-End. In *2005 IEEE Radio Frequency Integrated Circuits Symposium*, page 309.
- [59] P. Kapur H. Cho, K-H Koo and K.C. Saraswat. Performance Comparison between Cu/Low-L, Carbon Nanotube, and Optics for On-chip Global Interconnects, 2007. manuscript.

- [60] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff La-Coss, John Granacki, Jay Brockman, Apoorv Srivastava, William Athas, Vincent Freeh, Jaewook Shin, and Joonseok Park. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 57, New York, NY, USA, 1999. ACM.
- [61] Mark W. Hart. Step-and-Flash Imprint Lithography for Storage-Class Memory. http://www.molecularimprints.com/NewsEvents/tech_articles/new_articles/EIPBN2007.IBMv2.pdf, 2007.
- [62] Hisao Hayakawa, Nobuyuki Yoshikawa, Shinichi Yoroze, and Akira Fujimaki. Superconducting Digital Electronics. *Proc. of the IEEE*, 92(10), October 2004.
- [63] P. Hazucha, T. Karnik, J. Maiz, S. Walstra, B. Bloechel, J. Tschanz, G. Dermer, S. Hareland, P. Armstrong, and S. Borkar. Neutron Soft Error Rate Measurements in a 90-nm CMOS Process and Scaling Trends in SRAM from 0.25- μ m to 90-nm Generation. In *International Electron Devices Meeting*, pages 21.5.1–21.5.4, December 2003.
- [64] J. R. Heath, P. J. Kuekes, G. S. Snider, and R. S. Williams. A defect-tolerant computer architecture: Opportunities for nanotechnology. *Science*, 280:1716–1721, June 1998.
- [65] C. A. R. Hoare. Communicating sequential processes. In *Communications of the ACM*, volume 21, pages 666–677, 1978.
- [66] L. Hochstein, T. Nakamura, V.R. Basili, S. Asgari, M.V. Zelkowitz, J.K. Hollingsworth, F. Shull, J. Carver, M. Voelp, N. Zazworka, , and P. Johnson. Experiments to Understand HPC Time to Development. *Cyberinfrastructure Technology Watch Quarterly*, Nov. 2006.
- [67] T. Hogg and G. S. Snider. Defect-tolerant adder circuits with nanoscale crossbars. *IEEE Trans. Nanotechnol.*, 5(2):97–100, 2006.
- [68] T. Hogg and G. S. Snider. Defect-tolerant logic with nanoscale crossbar circuits. *JETTA*, 23(2-3):117–129, 2007.
- [69] D. Hopkins and et.al. Circuit techniques to enable a 430 gb/s/mm/mm proximity communication. In *IEEE International Solid State Circuits Conference*, pages 368–369, 2007.
- [70] H.Tanaka, M.Kido, K.Yahashi, M.Oomura, and et al. Bit cost scalable technology with punch and plug process for ultra high density flash memorye. pages 14–15. IEEE Symp. on VLSI technology, 2007.
- [71] Endicott Interconnect. HyperBGA Technology. <http://eitnpt1.eitny.com/contentmanager/literature/HYPERBGA.pdf>.
- [72] Mary Jane Irwin and John Shen, editors. *Revitalizing Computer Architecture Research*. Computing Research Association, December 2005.
- [73] iSuppli. isuppli market tracker, q3. <http://www.isuppli.com/catalog/detail.asp?id=8805>.
- [74] L. Jiang and et.al. Close-loop electro-osmotic micromechannel coolings system for VLSI circuits. *IEEE Trans. CPMT, Part A*, 25:347–355, September 2002.

- [75] Bill Joy and Ken Kennedy. *Information Technology Research: Investing in Our Future*. National Coordination Office for Computing, Information, and Communications, Arlington, VA, February 1999.
- [76] G. Y. Jung, S. Ganapathiappan, D. A. A. Ohlberg, D. L. Olynick, Y. Chen, W. M. Tong, and R. S. Williams. Fabrication of a 34 x 34 crossbar structure at 50 nm half-pitch by UV-based nanoimprint lithography. *Nano Letters*, 4:1225–1229, July 2004.
- [77] G. Y. Jung, E. Johnston-Halperin, W. Wu, Z. N. Yu, S. Y. Wang, W. M. Tong, Z. Y. Li, J. E. Green, B. A. Sheriff, A. Boukai, Y. Bunimovich, J. R. Heath, and R. S. Williams. Circuit fabrication at 17 nm half-pitch by nanoimprint lithography. *Nano Letters*, 6:351–354, March 2006.
- [78] P. Kapur and K.C. Saraswat. Optical Interconnections for Future High Performance Integrated Circuits. *Physica E*, 16:620–627, 2003.
- [79] B. Keeth and R. Baker. *DRAM Circuit Design: A Tutorial*. IEEE Press, 2000.
- [80] T. Kgil and et al. Picoserver: Using 3d stacking technology to enable a compact energy efficient chip multiprocessor. *ASPLOS*, 2006.
- [81] J. Kim, W. Dally, B. Towles, and A. Gupta. Microarchitecture of a high-radix router. In *IProceedings 32th Annual Int. Symp. on Computer Architecture(ISCA)*, pages 420–431, June 2005.
- [82] J.S. Kim, W.H. Cha, K.N. Rainey, S. Lee, and S.M. You. Liquid cooling module using FC-72 for electronics cooling. In *ITHERM '06*, pages 604–611, May 2006.
- [83] Graham Kirsch. Active Memory: Micron's Yukon. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 89.2, Washington, DC, USA, 2003. IEEE Computer Society.
- [84] David Koester. Exascale Study Application Footprints. <http://info.mitre.org/infoservices/selfserve/public.release.docs/2007/07-1449.pdf>, 2007.
- [85] P. Kogge. The EXECUBE Approach to Massively Parallel Processing. In *Int. Conf. on Parallel Processing*, Chicago, Aug. 1994.
- [86] P. Kogge. An Exploration of the Technology Space for Multi-Core Memory/Logic Chips for Highly Scalable Parallel Systems. In *IEEE Int. Workshop on Innovative Architectures*, pages 55–64, Oahu, HI, Jan. 2005.
- [87] J. Koomey. Estimating Total Power Consumption By Servers In The U.S. And The World, February 2007. Lawrence Berkeley National Laboratory, Final Report.
- [88] Christoforos Kozyrakis and David Patterson. Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 283–293, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.
- [89] P. J. Kuekes, G. S. Snider, and R. S. Williams. Crossbar nanocomputers. *Sci. Am.*, 293(5):72–80, 2005.

- [90] Sandia National Labs. Photo of Red Storm. <http://www.sandia.gov/NNSA/ASC/images/platforms/RedStorm-montage.jpg>.
- [91] J. H. Lee, X. Ma, D. B. Strukov, and K. K. Likharev. Cmol. pages 3.9–3.16, Palm Springs, CA, May 2005. NanoArch.
- [92] J.D Lee, S.H. Hur, and J.D. Choi. Effects of floating-gate interference on NAND flash memory cell operation. *IEEE Electron. Device Letters*, 23(5):pp. 264–266, May 2002.
- [93] Ana Leon, Jinuk Shin, Kenway Tam, William Bryg, Francis Schumacher, Poonacha Kongetira, David Weisner, and Allan Strong. A Power-Efficient High-Throughput 32-Thread SPARC Processor. pages 98–99, San Francisco, CA, 2006. ISSCC.
- [94] K. Likharev and D. Strukov. *CMOL: Devices, circuits, and architectures*, pages 447–478. Springer, 2005.
- [95] R. J. Luyken and F. Hofmann. Concepts for hybrid CMOS-molecular non-volatile memories. *Nanotechnology*, 14:273–276, February 2003.
- [96] X. Ma, D. B. Strukov, J. H. Lee, and K. K. Likharev. Afterlife for silicon: Cmol circuit architectures. pages 175–178, Nagoya, Japan, July 2005. IEEE Nanotechnol.
- [97] Junichiro Makino, Eiichiro Kokubo, and Toshiyuki Fukushima. Performance evaluation and tuning of GRAPE-6 towards 40 real Tflops. ACM/IEEE SC Conference, 2003.
- [98] M. Mamidipaka and N. Dutt. eCACTI: An Enhanced Power Estimation Model for On-chip Caches, 2004. Center for Embedded Computer Systems (CESC) Tech. Rep. TR-04-28.
- [99] J. Mankins. Technology Readiness Levels. <http://www.hq.nasa.gov/office/codeq/trl/trl.pdf>, April 1995.
- [100] John Markoff. Pentagon Redirects Its Research Dollars; University Scientists Concerned by Cuts in Computer Project. *New York Times*, pages section C, page 1, column 2, April 2005.
- [101] M. Masoumi, F. Raissi, M. Ahmadian, and P. Keshavarzi. Design and evaluation of basic standard encryption algorithm modules using nanosized complementary metal-oxide-semiconductor-molecular circuits. *Nanotechnology*, 17(1):89–99, 2006.
- [102] N. Nakajima and F. Matsuzaki, Y. Yamanashi, N. Yoshikawa, M. Tanaka, T. Kondo, A. Fujimaki, H. Terai, , and S. Yorozu. Design and implementation of circuit components of the SFQ microprocessor, CORE 1. *9th Int. Superconductivity Conf.*, 2003.
- [103] P. Mehrotra and P. Franzon. Optimal Chip Package Codesign for High Performance DSP. *IEEE Trans. Advanced Packaging*, 28(2), May 2005.
- [104] S.E. Michalak, K.W. Harris, N.W. Hengartner, B.E. Takala, and S.A. Wender. Predicting the number of fatal soft errors in Los Alamos National Laboratory’s ASC Q supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, September 2005.
- [105] John Michalakes, Josh Hacker, Richard Loft, Michael O. McCracken, Allan Snavely, Nicholas J. Wright, Tom Spelce, Brent Gorda, and Robert Walkup. WRF Nature Run. Int. Conf. for High Performance Computing, Networking, and Storage, nov 2007.

- [106] Micron. Micron system power calculator. <http://www.micron.com/support/designsupport/tools/powercalc/powercalc.aspx>.
- [107] N. Miura, H. Ishikuro, T. Sakurai, and T. Kuroda. A 0.14 pj/bit inductive coupling inter-chip data transceiver with digitally-controlled precise pulse shaping. In *IEEE International Solid State Circuits Conference*, pages 358–608, 2007.
- [108] Jos Moreira, Michael Brutman, Jos Castaos, Thomas Engelsiepen, Mark Giampapa, Tom Gooding, Roger Haskin, Todd Inglett, Derek Lieber, Pat McCarthy, Mike Mundy, Jeff Parker, and Brian Wallenfelt. Designing a Highly-Scalable Operating System: The Blue Gene/L Story. ACM/IEEE SC Conference, 2006.
- [109] R. Murphy, A. Rodrigues, P. Kogge, , and K. Underwood. The Implications of Working Set Analysis on Supercomputing Memory Hierarchy Design. Cambridge, MA, 2005. International Conference on Supercomputing.
- [110] Umesh Nawathe, Mahmudul Hassan, Lynn Warriner, King Yen, Bharat Upputuri, David Greenhill, Ashok Kumar, and Heechoul Park. An 8-core, 64-thread, 64-bit, power efficient SPARC SoC. San Francisco, CA, 2007. ISSCC.
- [111] John Nickolls. GPU Parallel Computing Architecture and the CUDA Programming Model. In *HotChips 19*, August 2007.
- [112] R. Numrich and J. Reid. Co-Array Fortran for parallel programming. In *ACM Fortran Forum 17, 2, 1-31.*, 1998.
- [113] Y. Okuyama, S. Kamohara, Y. Manabe, T. Kobayashi K. Okuyamaand K. Kubota, and K. Kimura. Monte carlo simulation of stress-induced leakage current by hopping conduction via multi-traps in oxide. pages 905–908. IEEE Electron. Devices Meeting, December 1998.
- [114] A.K. Okyay, D. Kuzum, S. Latif, D.A.B. Miller, and K.C. Saraswat. CMOS Compatible Silicon-Germanium Optoelectronic Switching Device, 2007. Manuscript.
- [115] A.J. Oliner, R.K. Sahoo, J.E. Moreira, and M. Gupta. Performance implications of periodic checkpointing on large-scale cluster systems. In *International Parallel and Distributed Processing Symposium*, April 2005.
- [116] D. R. Stewart P. J. Kuekes and R. S. Williams. The crossbar latch: Logic value storage, restoration, and inversion in crossbar circuits. *J. Appl. Phys.*, 97(3):034301, 2005.
- [117] R. Palmer, J. Poulton, W.J. Dally, J. Eyles, A.M. Fuller, T. Greer, M. Horowitz, M.Kellan, F. Quan, and F. Zarkesshvari. A 13 mw 6.25 gb/s transceiver in 90 nm cmos for serial chip-to-chip communication. In *IEEE International Solid State Circuits Conference*, pages 440–614, 2007.
- [118] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. <http://www.netlib.org/benchmark/hpl/>, 2004.
- [119] Daniel Reed, editor. *The Roadmap for the Revitalization of High-End Computing*. Computing Research Association, 2003.

- [120] K. Reick, P.N. Sanda, S. Swaney, J.W. Kellington, M. Floyd, and D. Henderson. Fault Tolerant Design of the IBM Power6 Microprocessor. In *HotChips XIX*, August 2007.
- [121] J. R. Reimers, C. A. Picconatto, J. C. Ellenbogen, , and R. Shashidhar, editors. *Molecular Electronics III*, volume 1006. Ann. New York Acad. Sci.
- [122] M. M. Ziegler C. A. Picconatto S. Das, G. Rose and J. E. Ellenbogen. *rchitecture and simulations for nanoprocessor systems integrated on the molecular scale*, pages 479–515. Springer, 2005.
- [123] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Nitya Ranganathan, Doug Burger, Stephen W. Keckler, Robert G. McDonald, and Charles R. Moore. TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP. *ACM Trans. Archit. Code Optim.*, 1(1):62–93, 2004.
- [124] B. Schroeder and G.A. Gibson. A Large-scale Study of Failures in High-performance-computing Systems. In *International Conference on Dependable Systems and Networks (DSN)*, pages 249–258, June 2006.
- [125] B. Schroeder and G.A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *USENIX Conference on File and Storage Technologies (FAST)*, February 2007.
- [126] Roy Schwitters and et al. Report on High Performance Computing for the National Security Community. <http://fas.org/irp/agency/dod/jason/ascii.pdf>, October 2003.
- [127] A. Shcham and K. Bergman. Building Ultralow-Latency Interconnection Networks Using Photonic Integration. *IEEE Micro*, 27(4):6–20, July-August 2007.
- [128] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on Soft Error Rate of Combinational Logic. In *International Conference on Dependable Systems and Networks (DSN)*, pages 389–398, June 2002.
- [129] Paul H. Smith, Thomas Sterling, and Paul Messina. *Enabling Technologies for Petaflops Computing*. MIT Press, 2005.
- [130] A. Snaveley, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A Framework for Application Performance Modeling and Prediction. pages 112–123, Baltimore, MD, November 2002. ACM/IEEE Conference on Supercomputing.
- [131] A. Snaveley, M. Tikir, L. Carrington, and E. Strohmaier. A Genetic Algorithms Approach to Modeling the Performance of Memory-bound Computations. Reno, NV, November 2007. ACM/IEEE Conference on Supercomputing.
- [132] Allan Snaveley, Larry Carter, Jay Boisseau, Amit Majumdar, Kang Su Gatlin, Nick Mitchell, John Feo, and Brian Koblenz. Multi-processor performance on the Tera MTA. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–8, Washington, DC, USA, 1998. IEEE Computer Society.
- [133] G. Snider. Computing with hysteretic resistor crossbars. *Appl. Phys. A-Mater. Sci. Process.*, 80(6):1165–1172, 2005.

- [134] G. Snider, P. Kuekes, T. Hogg, and R. S. Williams. Nanoelectronic architectures. *Appl. Phys. A-Mater. Sci. Process.*, 80(6):1183–1195, 2005.
- [135] G. Snider, P. Kuekes, and R. S. Williams. Cmos-like logic in defective, nanoscale crossbars. *Nanotechnology*, 15(8):881–891, 2004.
- [136] G. S. Snider and P. J. Kuekes. Nano state machines using hysteretic resistors and diode crossbars. *IEEE Trans. Nanotechnol.*, 5(2):129–137, 2006.
- [137] G. S. Snider and R. S. Williams. Nano/cmos architectures using a field-programmable nanowire interconnect. *Nanotechnology*, 18(3):035204, 2007.
- [138] L. Spainhower and T. A. Gregg. IBM S/390 Parallel Enterprise Server G5 fault tolerance: A historical perspective. *IBM Journal of Research and Development*, 43(5/6):863–874, 1999.
- [139] M. Stan, P. D. Franzon, S. C. Goldstein, J. C. Lach, and M. M. Ziegler. Molecular electronics: From devices and interconnect to circuits and architecture. *Proc. IEEE*, 91:1940–1957, November 2003.
- [140] Thomas Sterling. HTMT-class Latency Tolerant Parallel Architecture for Petaflops-scale Computation. <http://www.cs.umd.edu/users/als/NGS07/Presentations/8am-Sunday-Session/GaoSterling.pdf>, 1999.
- [141] D. B. Strukov and K. K. Likharev. Prospects for terabit-scale nanoelectronic memories. *Nanotechnology*, 16:137–148, January 2005.
- [142] D. B. Strukov and K. K. Likharev. Defect-tolerant architectures for nanoelectronic crossbar memories. *Journal of Nanoscience and Nanotechnology*, 7:151–167, January 2007.
- [143] F. Sun and T. Zhang. Defect and transient fault-tolerant system design for hybrid CMOS/-nanodevice digital memories. *IEEE Transactions on Nanotechnology*, 6:341–351, May 2007.
- [144] T. Sunaga, P. Kogge, and et al. A Processor In Memory Chip for Massively Parallel Embedded Applications. In *IEEE J. of Solid State Circuits*, pages 1556–1559, Oct. 1996.
- [145] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0, 2006. HP Labs Tech. Rep. HPL-2006-86.
- [146] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpfen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. *SIGARCH Comput. Archit. News*, 32(2):2, 2004.
- [147] IBM Blue Gene team. Overview of the IBM Blue Gene/P project. *IBM J. RES. & DEV.*, 52(1/2):199–220, 2008.
- [148] J. Tour. *Molecular Electronics*. World Scientific, Singapore, 2003.
- [149] S. Vangal and et al. An 80-Tile 1.28 TFLOPS Network-on-chip in 65 nm CMOS. pages 98–99, San Francisco, CA, 2007. ISSCC.
- [150] S. Vangal and et.al. An 80-tile 1.28 tflops network-on-chip in 65 nm cmos. In *IEEE International Solid State Circuits Conference*, pages 587–589, 2007.

- [151] Michael J. Voss and Rudolf Eigenmann. High-Level Adaptive Program Optimization with ADAPT. pages 93–102. Proc. of PPOPP’01, ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2001.
- [152] David Wallace. Compute Node Linux: Overview, Progress to Date and Roadmap. <http://www.nccs.gov/wp-content/uploads/2007/08/wallace.paper.pdf>, 2007.
- [153] T. Wang, Z. Qi, and C. A. Moritz. Opportunities and challenges in application-tuned circuits and architectures based on nanodevices. pages 503–511, Italy, April 2004. CCF.
- [154] W. Wang, M. Liu, and A. Hsu. Hybrid nanoelectronics: Future of computer technology. *J. Comp. Sci. Technol.*, 21(6):871–886, 2006.
- [155] J. Weinberg, M. O. McCracken, A. Snively, and E. Strohmaier. Quantifying Locality In The Memory Access Patterns of HPC Applications. Seattle, WA, November 2005. SC.
- [156] M. H. White, D. Adams, and J. Bu. On the go with sonos. In *IEEE Circuits and Devices*, volume 16, page 22, 2000.
- [157] Wikipedia. *Cray X-MP*.
- [158] S. Wilton and N. P. Jouppi. An Enhanced Access and Cycle Time Model for On-Chip Caches, 1994. DEC WRL Tech. Rep. 93/5.
- [159] V. Yalala, D. Brasili, D. Carlson, A. Huges, A. Jain, T. Kiszely, K. Kodandapani, A. Varadharajan, and T. Xanthopoulos. A 16-Core RISC Microprocessor with Network Extensions. pages 100–101, San Francisco, CA, 2006. ISSCC.
- [160] Katherine Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phillip Colella, and Alexander Aiken. Titanium: A High-Performance Java Dialect, *journal = Concurrency: Practice and Experience*. 10:825–836, 1998.
- [161] H.Y. Zhang, D. Pinjala, and T. Poi-Siong. Thermal Management of high power dissipation electronic packages: from air cooling to liquid cooling. In *EPTC 2003*, pages 620–625, 2003.
- [162] L. Zhang, J. Wilson, R. Bashirulla, L. Luo, J. Xu, and P. Franzon. A 32gb/s on-chip bus with driver pre-emphasis signaling. In *IEEE Custom Integrated Circuits Conference*, pages 773–776, 2006.
- [163] M. M. Ziegler, C. A. Picconatto, J. C. Ellenbogen, A. Dehon, D. Wang, Z. H. Zhong, and C. M. Lieber. Scalability simulations for nanomemory systems integrated on the molecular scale. *Molecular Electronics III*, 1006:312–330, 2003.
- [164] M. M. Ziegler and M. R. Stan. Cmos/nano co-design for crossbar-based molecular electronic systems. *IEEE Trans. Nanotechnol.*, 2(4):217–230, 2003.

Index

- 3D packaging, 131
- 3D stacking, 122
- fused multiply-add, 57
- activate, 117
- activity factor, 135
- adaptive mesh refinement, 66, 71
- Advanced Memory Buffer, 30
- aggressive strawman, 128, 175
- air conditioner, 144
- Air Vehicle Unstructured Solver, 66, 71, 75, 77, 78
- air-cooling, 164
- AIX, 36
- AMB, *see* Advanced Memory Buffer
- AMD K8, 28
- Amdahl's Law, 65, 227
- AMR, *see* adaptive mesh refinement
- anti-fuse, 109
- application performance, 7
- applicationa
 - cross-class, 11
- applications
 - Category I, 71, 72, 75, 79
 - Category II, 71, 72, 79, 80
 - Category III, 71, 72, 75, 79
 - Category IV, 72
- archival storage, 73, 127, 212
- array multi-core, 29
- ASCI, 23
- ASCI Red, 17
- aspect ratio, 106
- asynchronous, 44
- automatic parallelization, 40
- autotuners, 26
- Azul, 29
- Backus, 62
- balanced design, 6
- Ball Grid Arrays, 140
- bandwidth, 6, 113
 - bisection, 6, 69
 - checkpoint, 6
 - I/O, 6
 - local memory, 6
 - on-chip, 6
 - ratio, 7
 - requirements, 72
- BCH code, 108
- Beowulf, 34, 40
- Berkeley, 25
- BGA, 140
- bisection bandwidth, 6, 69, 74
- Bit Error Rate, 131, 217
- bit-line, 106, 116
 - capacitance, 106
- Brook-GPU, 28
- BTBB, 152
- bulk silicon, 93
- bussed interconnect, 130
- bytes to flops ratio, 7
- C*, 40
- C++, 152
- CACTI, 119
- CAF, 45
- capability computing, 7, 13, 79, 80
- capacity computing, 7, 12, 79
- Carbon Nanotubes, 137
- Cascade, 46
- Catamount, 36, 37
- Catamount System Libraries, 38
- Category I applications, 71, 72, 75, 79
- Category II applications, 71, 72, 75, 79, 80
- Category III applications, 71, 72, 75, 79
- Category IV applications, 72, 75
- Cell, 38
- cell modeling, 80
- cell power, 116, 119
- CFD:, 253

chalcogenide glass, 108
 challenges, 2, 209
 concurrency, 214
 energy, 209
 locality, 214
 memory, 212
 power, 209
 resiliency, 217
 storage, 212
 channel, 182
 Chapel, 46, 152
 charge pumps, 118
 checkpoint, 149, 184
 bandwidth, 6
 in Blue Gene/L, 149
 rollback, 71, 149
 chip junction temperature, 142
 chip level multi-processing, 28
 chip-level reliability, 147
 Cilk, 43, 152
 circuit switched network, 136
 circuit switching, 132
 clock, 55, 56
 Clock and Data Recovery, 134
 CM-2, 40, 41
 CMFortran, 40
 CMOL, 100
 CMP, 29
 CNL, 37
 CNT, 137
 Co-Array Fortran, 40, 45, 152
 collective network, 172
 collectives, 44
 communicating sequential processes, 39
 Compaq Himalaya, 149
 computational fluid dynamics, 69, 71
 computational rates, 5
 compute card, 170
 compute chip, 170
 compute node, 172
 Compute Node Linux, 37
 concurrency, 57
 concurrency challenge, 2, 214
 configurable logic, 137
 constant field scaling, 49
 constant voltage scaling, 49
 consumer-class disks, 122
 core, 175, 178
 CORE-1, 101
 COTS architectures, 23
 Cray
 Black Widow, 132
 MTA, 30
 XMP, 63
 XMT, 30
 XT, 31
 XT3, 38
 XTn, 39
 criticality, 219
 cross-class applications, 11
 crossbar, 100, 132
 crosspoint memory array, 120
 crosspoints, 137
 cryostat coolers, 101
 CSP, 39
 cubic scaling, 27
 CUDA, 28, 152
 current flow, 127
 custom architectures, 23
 Cyclops, 29, 31

 data center system, 8
 data center systems, 12, 13
 Data parallel languages, 40
 dataflow, 39
 Datastar, 63
 DDR, 113, 115
 DDR2, 29, 113, 115
 DDR3, 113, 115
 Defense Science Board, 21
 Delay Locked Loop, 130
 delivered power, 32
 departmental systems, 9, 14, 80
 dependability, 26
 device resiliency scaling, 147
 die stacking, 115
 die thinning, 115
 dielectric breakdown, 147
 DIMM, 29, 172
 direct chip-chip interconnect, 131
 direct immersion cooling, 143
 disk
 transfer time, 125
 capacity, 123
 consumer, 122
 drive reliability, 145

- enterprise, 123
- handheld, 123
- seek time, 125
- storage, 184
- technology, 122
- transfer rate, 125
- dissipated power, 32, 88
- distributed memory, 199
- DIVA, 31
- DLL, 130
- DNA, 97
- domino logic, 97
- dragonfly topology, 182
- DRAM, 106, 110, 212
 - activate mode, 117
 - capacitor, 116
 - chip architecture, 118
 - embedded, 107
 - fast cycle, 107
 - idle mode, 117
 - latency, 30
 - modes, 116
 - power, 115, 116
 - precharge mode, 117
 - read mode, 117, 118
 - reduced latency, 107
 - refresh mode, 117
 - reliability, 109
 - SER, 111
 - stacking, 115
 - Voltage Scaling, 116
 - write mode, 117, 118
- E/O conversion, 192
- E3 Initiative, 202
- E3SGS, 20
- Earth Simulator, 31, 41
- earthquake modeling, 79
- ECC, 111, 112, 147
- efficiency, 32, 54
- EIP, 153
- EIPs, 153
- electrical anti-fuses, 109
- electro-migration, 142, 147
- embarrassingly parallel, 72
- embedded DRAM, 107
- embedded systems, 10, 14, 34, 80
- enabling technologies, 23
- encoding for resiliency, 147
- endurance, 108
- energy, 210
 - and power challenge, 209
 - challenge, 2
 - efficiency, 95
 - per cycle, 88
 - per operation, 88, 89, 178, 210
 - scaling, 27, 178
- enterprise-class disks, 123
- environments, 198
- EOS/DIS, 74
- Exa Instructions Processed per second, 153
- exa-sized systems, 8
- exaflop, 9
- exaflops, 9
- Exascale, 9
- EXECUBE, 29, 31
- execution model, 39
- executive, 37
- external cooling mechanisms, 142
- fail-over, 149
- failure rate, 110, 145
- Failures in time, 110
- Fast Cycle DRAM, 107
- fault recovery, 149
- FB-DIMM, 30, 164
- Fe-RAM, 120
- featherweight supercomputer, 24
- feature size, 47, 104
- FeRAM, 120
- ferro-electric RAM, 120
- FG NAND, 109
- field programmable gate array, 98, 100
- field-programmable nanowire interconnect, 100
- file storage, 73, 162, 212
- fine line technologies, 141
- FINFET, 94
- FIT, 110, 145
- Flash memory, 107, 213
- flash rewrites, 213
- floating point unit, 175, 176
- floating trap layer, 108
- flop, 176
- flops, 5
- FLUX-1, 101
- flying bits, 111

Fortress, 47
 FPNI, 100
 FPU power, 210
 front-side bus, 30
 full swing signaling, 130
 fully depleted, 94
 fully depleted SOI, 93
 fuse, 112

 gap, 219
 GASNet, 47
 gate
 capacitance, 92
 dielectric, 92
 leakage, 92
 oxide, 92
 ghost node, 59
 ghost region, 41
 gigascale, 9, 17
 global channel, 183
 global interconnect, 72
 GPFS, 38
 GPU, 28, 152
 Grape, 7
 graphs, 71
 grid, 10
 group, 175, 183
 GUPS, 82, 212

 HAL, 37
 handheld-class disks, 123
 hard failures, 147
 Hardware Abstraction Layer, 37
 heating, ventilating, and air conditioning, 32
 heatsink, 165
 HECRTF, 23
 helper flip-flop, 118
 hierarchial multi-core, 29
 high performance CMOS, 89
 High performance Linpack benchmark, 9
 High Productivity Computing Systems, 17, 20, 22, 198
 high radix routers, 132
 high-end computing, 9
 high-K gate dielectric, 92
 high-performance computing, 9
 Holographic memory, 126
 HPC, 9

 HPCMO test case, 63
 HPF, 39–41
 HPL, 9, 69, 75, 77
 HPUX, 36
 hurricane forecast, 78
 HVAC, 32
 hybrid logic, 97
 hybrid technology, 100
 Hybrid Technology Multi-Threaded, 17, 100
 HYCOM, 72
 Hycom, 66
 HyperBGA, 140

 I/O bandwidth, 6
 IBM
 Blue Gene, 31, 38, 39, 63, 170
 Blue Gene/L, 63, 149
 Cell, 30
 Cyclops, 29
 G5, 149
 Power, 39
 Power 4, 63
 Power 6, 34
 Power6, 149
 ILP, 56
 imprint lithography, 121
 Infiniband, 31
 innovation trends in programming languages, 151
 instruction level parallelism, 56
 instructions per second, 5
 interconnect, 3, 127
 bus, 130
 direct chip-chip, 131
 energy loss, 127
 off-chip wired, 130
 on-chip and wired, 130
 studs, 141
 switched, 132
 time loss, 127
 internal cooling mechanisms, 142
 intrinsic delay, 89
 ionizing radiation, 111
 IPS, 5
 Irvine Sensors, 141
 Itanium, 29, 34
 ITRS, 47

 JAVA, 47

- Java, 152
- JEDEC, 113
- JJ, 100
- John Backus, 62
- Josephson Junction, 100
- Kiviat diagrams, 61
- L1 cache, 29
- LAPI, 47
- laser drilled vias, 141
- laser-trimmed fuses, 109
- latency, 113, 127
- latency requirements, 72
- Level 1 Packaging, 139
- Level 1 packaging, 140
- Level 2 Packaging, 139
- Level 2 packaging, 141
- Level 3 Packaging, 139
- life cycle of programming languages, 152
- lightweight kernel, 199
- link card, 172
- Linpack, 53
- Linux, 36
- liquid cooling, 166, 173
- Lisp, 40
- list search, 82
- Little's Law, 72, 77
- load balancing, 199
- Loadleveler, 38
- local channel, 183
- local memory bandwidth, 6
- locales, 46
- locality, 68, 69, 75
- locality challenge, 2, 214
- locality clusters, 72
- locality-aware architectures, 226
- logic
 - hybrid, 97
 - low power, 89
 - nonsilicon, 97
- low operating power CMOS logic, 89
- low swing interconnect, 130
- low voltage signaling, 180
- LS-DYNA, 10
- Lustre, 38
- Luxterra, 135
- Mach-Zender modulator, 134
- macrokernel, 37
- magnetic disks, 122
- magnetic RAM, 120
- Magnetic Random Access Memory, 109
- magnetic tunnel junctions, 109
- main memory, 5, 212
- main memory power, 211
- mantle physics, 79
- manycore, 24, 26, 28
- Maspar, 40, 41
- Maui scheduler, 38
- mean time to interrupt, 145
- membrane modeling, 79
- memory, 212
 - bandwidth, 113, 115
 - bank, 118, 157
 - capacity, 59
 - cell power, 116
 - challenge, 2, 212
 - consistency, 152
 - contoller, 30
 - footprint, 72
 - hierarchy, 103
 - intensive applications, 66
 - latency, 113
 - main, 5
 - management, 36
 - mat, 118
 - module, 115
 - packaging, 114
 - power, 115, 119
 - requirements, 72
 - socket reliability, 111
 - sub-bank, 118
 - wall, 18, 35, 103, 216
- Merrimac, 178
- message passing, 39, 172
- metadata, 73, 127
- metrics, 5
- micro-FBGA, 115
- micro-FBGA package, 115
- microkernel, 37
- Middleware, 38
- midplane, 172
- minimal routing, 183
- MLC, 107
- MODFET, 135
- module level cooling, 142

- molecular logic, 97
- molecular switches, 100
- Moore's Law, 63, 65
- MPI, 39, 44, 152
- MPI-2, 44
- MPICH, 44
- MRAM, 102, 109, 120
- MTJ, 109
- MTTI, 145
- multi-core, 24, 25, 28, 31, 91
- multi-level cell, 107
- multi-threading, 28, 30, 43
- multiple drug interactions, 79
- mux/demux, 121
- Myrinet, 31

- NAND flash, 107
- NAND memory, 109
- Nano-enabled Programmable Crosspoints, 137
- nanomemory, 213
- nanopositioners, 135
- nanowire, 100
- national security applications, 22
- nature run, 79
- new applications, 13
- Niagara, 29, 30, 34
- NoC, 192, 193
- node, 164, 172, 175, 181
- non silicon logic, 97
- non-minimal routing, 184
- non-volatile memory, 107, 120
- nonvolatile switches, 98
- north bridge, 30
- NT, 37
- NVRAM, 120

- O/E conversion, 192
- object-oriented programming, 152
- OCM, 196
- off-chip wired interconnect, 130
- on-chip
 - access, 179
 - bandwidth, 6
 - data transport, 180
 - interconnect, 28
 - wired interconnect, 130
- open row, 120
- OpenMP, 40, 42
- OpenMPI, 44
- operating environment, 35, 150, 198
- Opteron, 34
- optical
 - alignment, 135
 - circuit switch, 136
 - interconnection, 191
 - logic, 97
 - modulators, 134, 136
 - MOSFETs, 136
 - packet switch, 136
 - router, 192
- optical receiver, 135
- optically connected modules, 196
- OS kernels, 151
- out of core algorithms, 73
- out of order execution, 28
- overall concurrency, 55
- Overflow, 69

- p-threads, 151
- packaging, 114, 139
 - Level 1, 139, 140
 - Level 2, 139, 141
 - Level 3, 139
- packet switched network, 136
- packet switching, 132
- page-oriented memory, 126
- pages, 36
- parallelism, 25, 54, 55
- parity, 147
- partially depleted, 94
- partially depleted SOI, 93
- Partitioned Global Address Space, 45
- PBS, 38
- PCB, 141
- PCRAM, 108, 120
- PDE, 71
- PDU, 32, 33
- peak bandwidth, 127
- peak performance, 17
- persistent storage, 6
- persistent surveillance, 11, 25
- peta-sized systems, 8
- petaflops, 17
- Petascade, 9, 17, 80
- PGAS, 45
- pGAS, 32, 40, 152, 172, 206

phase change memory, 108
 phase change RAM, 120
 Phased Locked Loop, 130
 photonic interconnect, 192
 photonics, 23
 physical attributes, 6
 PIM, 31
 pipelined multi-core, 29
 pipelining, 113, 114
 PITAC, 21
 PLA, 100
 PLL, 130
 point-to-point interconnect, 130
 popular parallel programming, 24
 POSIX API, 43
 power, 88, 210
 challenge, 2
 delivered, 32
 density, 88, 89
 distribution, 149
 efficiency, 32
 wall, 35, 53
 Power Distribution Unit, 32
 Power Supply Unit, 32
 power-adaptive architectures, 227
 precharge, 117
 Printed Circuit Board, 141
 process control thread, 37, 38
 process management, 36
 Processing In Memory, 31
 processor channel, 183
 processor chip, 175
 program synchronization, 152
 programmable crosspoints, 137
 programmable logic array, 100
 programmable redundancy, 109
 programming languages
 innovative path, 151
 life cycle, 152
 road map, 152
 standardization path, 152
 standardization trends, 152
 programming model, 38
 property checking for resiliency, 148
 protein folding, 79
 prototyping phase, 3
 PSU, 32
 PThreads, 43
 Pthreads, 152
 PVFS, 38
 Q. Kernel, 37
 QCA, 97
 Quadrics, 31
 quantum cellular automata, 97
 quantum computing, 97
 quantum well modulators, 136
 quilt packaging, 181
 quintessential kernel, 37
 RA, 69
 rack, 175, 182, 184
 radar plots, 61
 radiation hardness, 100, 109, 111
 Random Access, 69
 Rapid Single Flux Quantum, 100, 222
 RAW, 29, 31
 read-write cycles, 108
 real-time, 7, 13
 recommendations, 2
 red shift, 62
 Red Storm, 9, 31, 36, 164
 Reduced Latency DRAM, 107
 reduced latency DRAM, 113
 reductions, 42
 redundancy, 112
 Reed-Solomon code, 108
 refresh time, 110
 register files, 97
 reliability, 110, 112
 DRAM, 109
 socket, 111
 Reliability, Maintainability, and Availability, 111
 remote memory accesses, 5
 replication for resiliency, 148
 research agenda, 3
 research thrust areas, 218
 resiliency, 144, 217
 encoding, 147
 sparing, 148
 causes, 147
 challenge, 2, 217
 property checking, 148
 replication, 148
 scrubbing, 148
 techniques, 147

resistive memory, 137
 resistive RAM, 120
 retention time, 110, 112, 114
 RFCTH, 69
 ring modulator, 134
 RLDRAM, 113
 RMA, 111
 Rmax, 53
 road map for programming languages, 152
 roll-back, 149
 router chip, 183, 184
 routers, 132
 Rpeak, 53
 RRAM, 120

 Sandia National Labs, 9
 SATA, 184
 scheduler, 36
 SCI, 36
 SCM, 222
 SCP, 141
 scratch storage, 6, 73, 162, 212
 scrubbing for resiliency, 148
 SDRAM, 115
 Seastar, 31
 SECDDED, 147
 seek time, 125
 seek times, 214
 sense-amp, 106, 111, 118
 SER, 106, 111
 DRAM, 111
 SRAM, 111
 SERDES, 130, 220
 server blades, 18
 server systems, 17
 SEU, 145, 147, 218
 shared memory model, 41
 signal to noise ratio, 131
 signalling on wire, 130
 signalling rate, 127
 silicon carrier, 140, 141
 Silicon on Insulator, 93, 94
 silicon photonic integration, 134
 SIMD, 39–41, 152, 179
 Single Chip Package, 141
 single event upset, 145, 147, 218
 single mode waveguides, 134
 slab allocator, 36

 SLC Flash, 108
 slice, 3
 SMP, 31, 34
 SNIC, 222
 SNR, 131
 socket, 145, 165
 soft error rate, 106, 111
 SOI, 92–94, 134
 Solaris, 36
 solder bump, 144
 SONOS memory, 108, 109
 sparing for resiliency, 148
 spatial locality, 68
 speedup, 61
 spintronics, 23
 SPMD, 45
 sPPM, 80
 SRAM, 97, 106
 SRAM SER, 111
 SSTL, 30
 stacked cell DRAM, 106
 static RAM, 97, 106
 storage
 capacity, 5
 challenge, 2, 212
 persistent, 6
 scratch, 6
 Storm-1, 30
 strained silicon, 92
 strawman-aggressive, 175
 STREAM, 69
 stream processors, 30
 structured grids, 71
 sub-threshold leakage, 92, 93
 sub-threshold slope, 93, 94
 Sun Niagara, 29, 30, 132
 super-cores, 192
 super-pipelining, 89
 supercomputing, 9, 31
 superconducting, 23
 supply voltage scaling, 94
 sustainable bandwidth, 127
 sustained performance, 17, 150
 switched interconnect, 130, 132
 synchronous, 44
 system architecture phase, 3
 System Call Interface, 36
 systolic, 39

taper, 179
 TEC, 136, 142
 technology demonstration phase, 3
 temperature, 147
 temporal locality, 68
 tera-sized systems, 8
 teraflop, 17
 teraflops, 17
 Teraflops Research Chip, 29, 31
 Terascale, 9, 14, 17
 thermal electric coolers, 136
 thermal electric cooling, 142
 thermal resistance, 142
 thermal stress, 147
 thread level concurrency, 55, 56
 threads, 43, 199
 threshold voltage, 92, 95, 147
 Through Silicon Vias, 131, 140
 through wafer vias, 115
 thrust areas, 218
 tipping point, 20
 Titanium, 152
 TLC, 55, 56
 TMR, 149
 Top 500, 53
 top 500, 53
 total concurrency, 57
 transactional memory, 29, 152
 transfer rate, 125
 transfer time, 125
 transistor density, 88
 transport model, 92
 trench cell DRAM, 106
 tri-gate transistors, 94
 triple modular redundancy, 149
 TRIPS, 31
 Turing lecture, 62
 two phase cooling, 143

 UltraSparc, 34
 Unified Parallel C, 45
 Uninterruptible Power Supply, 32
 unstructured grids, 71
 UPC, 40, 45, 152
 UPS, 32, 33
 upscaling, 13

 variability, 147

Variable Retention Time, 111
 VCSEL, 134
 Vdd, 89
 vector, 40
 vertical-cavity surface-emitting laser, 134
 via, 115
 Virtual File System interface, 36
 virtualization, 151
 visualization, 74
 VLIW, 179
 Voltage Regulator, 32
 von Neumann, 35
 von Neumann bottleneck, 62
 VR, 32
 VRT, 111, 112

 weak scaling, 71, 77
 wearout, 147
 Windows NT, 37
 Windows Server 2003, 37
 wordline, 118
 working sets, 72
 WRF, 63, 66, 69, 71, 75, 77–79

 X10, 47, 152
 X10q, 29
 Xelerator X10q, 29
 Xeon, 34

 YARC router, 132
 YUKON, 31

 zettaflops, 20
 ZPL, 46

APPENDIX B

FINAL REPORT OF EXASCALE COMPUTING SOFTWARE STUDY

ExaScale Software Study: Software Challenges in Extreme Scale Systems



**Saman Amarasinghe
Dan Campbell
William Carlson
Andrew Chien
William Dally
Elmootazbellah Elnohazy
Mary Hall
Robert Harrison
William Harrod
Kerry Hill
Jon Hiller
Sherman Karp
Charles Koelbel
David Koester
Peter Kogge
John Levesque
Daniel Reed
Vivek Sarkar, Editor & Study Lead
Robert Schreiber
Mark Richards
Al Scarpelli
John Shalf
Allan Snively
Thomas Sterling**



September 14, 2009

**The views expressed are those of the authors and do not reflect the
official policy or position of the Department of Defense or the U.S. Government.**

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ECSS Report

Exascale Software Study: Software Challenges in Extreme Scale Systems

ECSS Report

DISCLAIMER

The material in this document reflects the thoughts and opinions of the participants only, and not those of any of the universities, corporations, or other institutions to which they are affiliated.

The views, opinions, and/or findings contained in this article are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

FOREWORD

This document reflects the thoughts of a group of highly talented individuals from universities, industry, and government research labs on the software challenges that will need to be addressed for the Extreme Scale systems that are anticipated in the 2015 – 2020 time-frame. It was drawn from a study conducted over a series of seven meetings held from June 2008 to February 2009. The goal of the study was to examine the state of the art, identify key challenges, and outline elements of a technical approach that can address the challenges without prescribing specific solutions. The report was assembled from input provided by study participants and guests over this short period of time. As such, all inconsistencies reflect either misunderstandings by the editor or areas where there were difference of opinion among members of the team. There was, however, unanimous agreement about the key challenges that surfaced from the study, and the criticality of addressing the software challenges in conjunction with hardware challenges when developing future Extreme Scale systems.

I am honored to have been part of this study, and wish to thank the study members and guests for their dedication to the field of parallel software and systems, and for all their hard work in contributing to the study.

Vivek Sarkar, Editor and Study Lead
Rice University
September 14, 2009.

September 14, 2009

ii

ECSS Report

Contents

1	Executive Summary	1
2	Exascale Hardware Characterization	3
2.1	Strawmen	3
2.2	The Aggressive Strawman Architecture	5
2.3	The Summary Extrapolations	8
3	Extreme Scale Software Execution Models and Metrics	10
3.1	Execution Models	10
3.2	Metrics	14
4	Challenges in Developing Applications for Extreme Scale Systems	16
4.1	Application Overview	16
4.2	Application Scaling	18
4.3	Emerging Extreme Scale Applications	22
4.4	“Traditional” HPC Applications at Exascale	22
4.5	Coupled Models	23
4.6	Exascale Data Intensive and Data Mining Applications	25
4.7	Real-time Departmental Extreme Scale Applications	29
4.8	Framework Technology	30
4.9	Footprints	39
4.10	Two Illustrative Graph Scenarios	41
5	Challenges in Expressing Parallelism and Locality in Extreme Scale Software	45
5.1	Application Programming for Extreme Scale Require Fundamental Breakthroughs	45
5.2	Portable Expression of Massive Parallelism	46
5.3	Portable Expression of Locality	47
5.4	Portable Expression of Synchronization with Dynamic Parallelism	50
5.5	Support for Composable and Scalable Parallel Programs with Algorithmic Choice	51
5.6	Managing Heterogeneity in a Portable Manner	51
6	Challenges in Managing Parallelism and Locality in Extreme Scale Software	53
6.1	Operating System Challenges	53
6.2	Runtime Challenges	57
6.3	Compiler Challenges	61
6.4	Library Challenges	63

7	Challenges in Supporting Extreme Scale Tools	67
7.1	History of Tools and Development Environments	67
7.2	Overview of Extreme Scale Development Environment Challenges	68
7.3	Enabling Technologies for Exascale Tools	70
7.4	Scenarios for Interaction with Tools	72
7.5	Summary	77
8	Technical Approach	78
8.1	Software-Hardware Interfaces in an Extreme Scale System	78
8.2	Opportunities for Software-Hardware Co-Design	81
8.3	Deconstructed Operating Systems	84
8.4	Global OS and Self-Aware Computing	87
8.5	Silver: An Example Execution Model and Technical Approach for Extreme Scale Systems	90
9	Conclusions	95
A	Additional Extreme Scale Software Ecosystem Requirements	97
A.1	Real-time and Other Specialized Requirements in Embedded Software	97
A.2	Tools and Development Environments	100
B	Definitions of Seriality, Speedup, and Scalability	105
B.1	Definitions	105
B.2	Approximate Inter-relationships	106
B.3	Algorithmic Scalability	107
B.4	Speedup	111
B.5	Efficiency	114
B.6	More Nuanced Views of Speedup	116
B.7	Memory and Bandwidth Scaling	118
B.8	Relevance to Exascale	123
C	CUDA as an Example Execution Model	127
D	Extreme Scale Software Study Group Members	130
D.1	Committee Members	130
D.2	Biographies	131
E	Extreme Scale Software Study Meetings, Speakers, and Guests	139

Chapter 1

Executive Summary

This report presents the findings and recommendations of the **Exascale Software Study** conducted from June 2008 to February 2009. A characterization of Extreme Scale systems can be found in the recent report on “Technology Challenges in Achieving Exascale Systems” [62]. This characterization identifies three distinct classes of systems:

- **Data-center-sized Exascale systems**, capable of delivering 1 ExaFlops or 1 ExaOps¹, which is 1,000× the capability of currently emerging Petascale data-center-sized systems.
- **Departmental-sized Petascale systems** that allow the capabilities of a Petascale system to be shrunk in size and power to fit within a few racks, allowing widespread deployment.
- **Embedded Terascale systems** that reduce Terascale capability to a few chips and a few ten’s of watts, thereby enabling deployment in a range of embedded environments.

Since the first system class listed above achieves Exascale performance, the terms *Exascale* and *Extreme Scale* are often interchangeably in the community and in this report. However, whenever possible, we prefer to use *Extreme Scale* to refer to systems across all three classes and *Exascale* specifically for the largest data-center sized system class.

The focus of this study is on *software challenges for Extreme Scale systems*. The scope of software considered spans the spectrum of operating systems; runtimes for scheduling, memory management, communication, performance monitoring, power management, and resiliency; computational libraries; compilers; programming languages; and application frameworks. Though there are significant differences in the software requirements for the three classes of Extreme Scale systems, all of them share some critical challenges: they will be built using *massive multi-core processors* with 100’s of cores per chip, their performance will be *driven by parallelism and constrained by energy*, and they will be subject to *frequent faults and failures*. Thus, the three key challenges for Extreme Scale software are *Concurrency*, *Energy Efficiency* and *Resiliency*. This study addresses the Concurrency and Energy Efficiency challenges, whereas the third challenge is addressed by a companion study on Exascale Resiliency. Development of Extreme Scale algorithms and applications as well as development of Extreme Scale hardware are outside of the scope of this study. However, identification of opportunities for software-hardware co-design, as well interfaces between applications and system software and between system software and hardware, are very much in scope.

The concurrency challenge is manifest in the need for software to expose at least 1000× more concurrency in applications for Extreme Scale systems, relative to current systems. It is further

¹Following common usage, “ops” refers to operations per second in this report unless otherwise specified.

exacerbated by the projected memory-computation imbalances in Extreme Scale systems, with Bytes/Ops ratios that may drop to values as low as 10^{-2} where Bytes and Ops represent the main memory and computation capacities of the system respectively. These ratios will result in $100\times$ reductions in memory per core relative to Petascale systems, with accompanying reductions in memory bandwidth per core. Thus, a significant fraction of software concurrency in Extreme Scale systems must come from exploiting more parallelism within the computation performed on a single datum *i.e.*, from strong scaling or from the “new-era” weak scaling discussed in Chapter 4. Strong scaling often involves more frequent communication and synchronization than weak scaling, which in turn contributes to the energy efficiency challenge since data movement and synchronization are major contributors to energy costs in Extreme Scale systems. Another major obstacle to achieving a large degree of concurrency arises from the serialization bottlenecks in current system software approaches to communication and synchronization. A new software stack can reduce these overheads by orders of magnitude, especially with software-hardware co-design, thereby making it possible to achieve the parallel efficiency needed for Extreme Scale systems.

The energy efficiency challenge is critical also because all three classes of Extreme Scale systems will be expected to deliver their $1000\times$ improvements in computation capability while essentially remaining within the power budgets of current systems. An aggressive hardware design for data-center-sized systems will need at least 60MW of power to achieve an Exa-op level of performance, under highly idealized zero-overhead assumptions for software [62]. When current software overheads are taken into account, it is clear that the Exascale capability cannot be achieved without a significant redesign of the system software stack.

As discussed in this report, current software approaches will be inadequate in enabling future Grand Challenge applications on Extreme Scale systems. Instead, the potential for a $1000\times$ increase in computation capability offered by each class of Extreme Scale system will only be achievable through radical re-design of the underlying execution model and system software and hardware. Current execution models and system designs won’t work at Extreme Scale because of their sequential foundations and their inherent energy inefficiencies. In addition, any attempt to use current execution models at Extreme Scale will result in prohibitively large costs in programmability. Recent trends in High Productivity Computing Systems (HPCS) have demonstrated reductions in the human effort required to develop high-productivity software for current Petascale systems, but they do not address the requirements of Extreme Scale architectures such as energy-constrained many-core parallelism and heterogeneous processors. Also, while there is some overlap between system software requirements for Extreme Scale and those for large scale commercial data centers, there are also significant differences. Commercial system software for Cloud Computing is primarily focused on optimizing throughput capacity of independent jobs, whereas system software for Extreme Scale must be capable of delivering a $1000\times$ increase in parallelism to a single job.

This report recommends a technical approach for developing new software stacks for future Extreme Scale systems that includes new execution models and metrics (Chapter 3), with a focus on new implementations of grand challenge applications (Chapter 4) either developed from scratch or scaled up from existing Petascale applications. An Extreme Scale software stack must enable parallelism and locality to be expressed at the finest granularities possible so as to support *forward scalability* (Chapter 5), low overhead management of parallelism and locality (Chapter 6), and integration with future tools (Chapter 7). Finally, the new software stacks must tightly integrated with new Extreme Scale hardware via a rich set of hardware API interfaces in support of software-hardware co-design (Chapter 8). A 12-page abbreviated version of this report is available in [120].

Chapter 2

Exascale Hardware Characterization

The objectives of the prior exascale systems study [62]. were to understand the course of mainstream computing technology, and determine whether or not it would allow a 1,000X increase in the computational capabilities of computing systems by the 2015 time frame. As mentioned earlier, the focus was on the technology to address three classes of systems: data center, departmental, and embedded.

This chapter summarizes the hardware configurations from which the challenges were identified, and upon which much of the software discussion in this report is premised. As a summary, Table 2.1 lists the characteristics of the “aggressive strawman” which drove many of the prior reports conclusions. While the prior study focused on Flops as a primary measure of computation, most of the issues discussed in this chapter are also applicable to benchmarks that are not floating-point intensive.

2.1 Strawmen

To understand these challenges, we not only surveyed the technology space, but also did an extrapolation from three different baselines of potential approaches for achieving the data center scaled system. These “strawmen” included:

- An evolutionary approach, termed a “heavyweight strawman” that assumed machines that used commodity high performance microprocessors on relatively large, high heat dissipating, circuit boards, with separate routing and memory chips. A few dozen such cards would make up a computing “rack.” Examples of such systems today include the Red Storm machine and its commercial XT derivatives from Cray, Inc.
- A second evolutionary approach, termed a “lightweight strawman” that assumed customized, lower power, microprocessors that permitted integration of a complete memory plus processing node on a small circuit card that could be stacked by the thousands into bigger systems. The IBM Blue Gene series is typical of this class today.
- A “clean sheet of paper” approach, termed the “aggressive strawman,” that still assumed silicon for its base technology, but one where the transistor parameters could be adjusted for maximum delivered performance per unit of energy consumed.

Table 2.2 lists some of the salient characteristics of both the two strawmen based on today’s machine architectures, and the aggressive strawman system as it might exist in 2015. In this table,

Exascale Software Study

Characteristic	Exascale System Class				
	Exaflops Data Cen- ter	20 MW Data Cen- ter	Department	Embedded A	Embedded B
Top-Level Attributes					
Peak Flops (PF)	9.97E+02	303	1.71E+00	4.45E-03	1.08E-03
Cache Storage (GB)	3.72E+04	11,297	6.38E+01	1.66E-01	4.03E-02
DRAM Storage (PB)	3.58E+00	1	6.14E-03	1.60E-05	1.60E-05
Disk Storage (PB)	3.58E+03	1,087	6.14E+00	0	0
Total Power (KW)	6.77E+04	20,079	116.06	0.290	0.153
Normalized Attributes					
GFlops/watt	14.73	14.73	14.73	15.37	7.07
Bytes/Flop	3.59E-03	3.59E-03	3.59E-03	3.59E-03	1.48E-02
Disk Bytes/DRAM Bytes	1.00E+03	1.00E+03	1.00E+03	0	0
Total Concurrency (Ops/ Cycle)	6.64E+08	2.02E+08	1.14E+06	2968	720
Off-chip Memory Band- width (B/sec per flops)	0.0025	0.0025	0.0025	0.0025	0.01
Off-chip Network Band- width (B/sec per flops)	0.0008	0.0008	0.0008	0.0008	0.0032
Component Count					
Cores	1.66E+08	50,432,256	2.85E+05	742	180
Microprocessor Chips	223,872	67,968	384	1	1
Router Chips	223,872	67,968	384	0	0
DRAM Chips	3,581,952	1,087,488	6,144	16	16
Total Chips	4,029,696	1,223,424	6,912	17	17
Total Disk Drives	298,496	90,624	512	0	0
Total Nodes	223,872	67,968	384	1	1
Total Groups	18,656	5,664	32	0	0
Total racks	583	177	1	0	0
Connections					
Chip Signal Contacts	8.45E+08	2.57E+08	1.45E+06	2,752	2,752
Board connections	1.86E+08	5.65E+07	3.19E+05	0	0
Inter-rack Channels	2.35E+06	7.14E+05	8,064	0	0

Table 2.1: Exascale class system characteristics derived from aggressive design.

Exascale Software Study

	Heavyweight Strawman based on XT4 (today)	Lightweight Strawman based on BG/P (today)	Aggressive Strawman (in 2015 technology)
Cores per socket	2	4	742
Sockets per node	4	1	12
Nodes per rack	24	1024	32
Racks per system	120	16	583
Cores per system	23,040	65,536	166,113,024
Sockets per system	11,520	16,384	223,872
Nodes per system	2,880	16,384	18,656
Flops per cycle per core	2	4	4
Clock (GHz)	2.6	0.85	1.55
GFlops per second per core	5.2	3.4	6.2
GFlops per second per socket	10.4	13.6	4,600
GFlops per second per node	41.6	13.6	55,205
GFlops per second per rack	998.4	13,926.4	1,766,554
GFlops per second per system	119,808.0	222,822.4	1,029,900,749
GBytes per core	2	0.5	0.0216
GBytes per socket	4	2	16
GBytes per node	16	2	192
GBytes per rack	384	2,048	6,144
Gbytes per system	46,080	32,768	3,581,952
Bytes per flop per second	0.3846	0.1471	0.0035

Table 2.2: Characteristics of hardware baselines.

a “core” is a processor capable of independent execution of a program thread, a “socket” is a single microprocessor chip, a “node” is a combination of processing, memory, and routing capable of complete program execution, and a “rack” is a refrigerator-sized enclosure that contains some number of nodes.

2.2 The Aggressive Strawman Architecture

The aggressive strawman was an attempt to see how far we could push if we started with a clean sheet of paper, especially to maximize gigaflops per watt. Figure 2.1 summarizes the resulting architecture of such a system. We assume a 2013 technology node of 32 nm silicon as a baseline for the projection, but with an alternative set of process parameters that permits much more aggressive voltage scaling than would be possible with “logic as usual” in the same time frame.

Figure 2.1 starts with a Floating Point Unit (**FPU**) along with its register files and amortized instruction memory. Four FPUs along with instruction fetch and decode logic and an L1 data memory forms a **core**. We combine 742 such cores on a 4.5Tflops, 150W active power (215W total) **processor chip**. This chip along with 16 1GB DRAMs forms a **Node** with 16GB of memory capacity. Together this combination by itself corresponds to an aggressive embedded exascale system. Using alternative silicon process parameters allows us to project about a factor of three in energy savings per flop over the conventional process.

The final three groupings correspond to the three levels of bigger system interconnection. 12

September 14, 2009

Page 5

ECSS Report

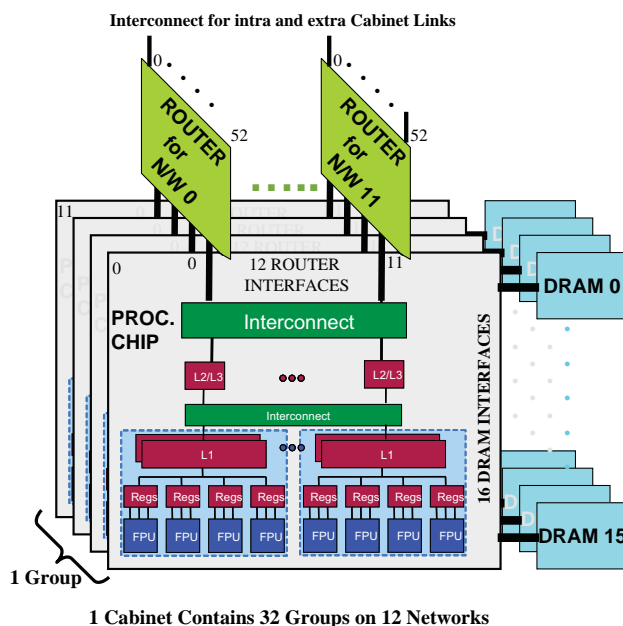


Figure 2.1: Aggressive strawman architecture.

nodes plus 12 separate routing chips form a **group**, 32 groups are packaged in a **rack**, and 583 **racks** are required to achieve a peak of 1 exaflops. We also assume 16 disk drives per group for secondary storage (192 TB in 2015), or an equivalent 512 drives (6.1 PB) per rack.

One rack by itself corresponds to over a petaflop, and thus corresponds to a departmental exascale system.

2.2.1 Adaptively Balanced Node

One of the concerns with the original strawman baseline of column C of Table 2.2 was that the bandwidth from the 742-core chip to the local memory was on the order of 0.0025 bytes per flop, and 0.00076 bytes per flop to off node memory. This is orders of magnitude less than what is typically considered desirable, but all that could be supported if one wanted in some sense to spend about equal power across processing, memory access, and interconnect.

To possibly overcome this on an application-by-application basis, the prior report [62] also proposed in Section 7.3.7 an *adaptive node design* i.e., a design that would support the maximum dissipative power to be spent in **either** processing, memory, or interconnect by themselves, and then allow some power-adaptive mechanisms to select what mix of all three does best for a particular application or application phase, and without exceeding the power dissipation limits of the chips.

The resulting design point would hold 1060 cores which when all at full speed would consume the total chip power, and provide 6.4 Tflops per chip, with no memory or network bandwidth. Alternatively, this adaptive design also widened the memory interfaces to provide an order of magnitude more bandwidth, if all one wanted to do was access memory. For applications which need some of both processing and memory bandwidth, intermediate points could be chosen that are balanced from a performance perspective, and utilize the available power capabilities to best efficiency.

Exascale Software Study

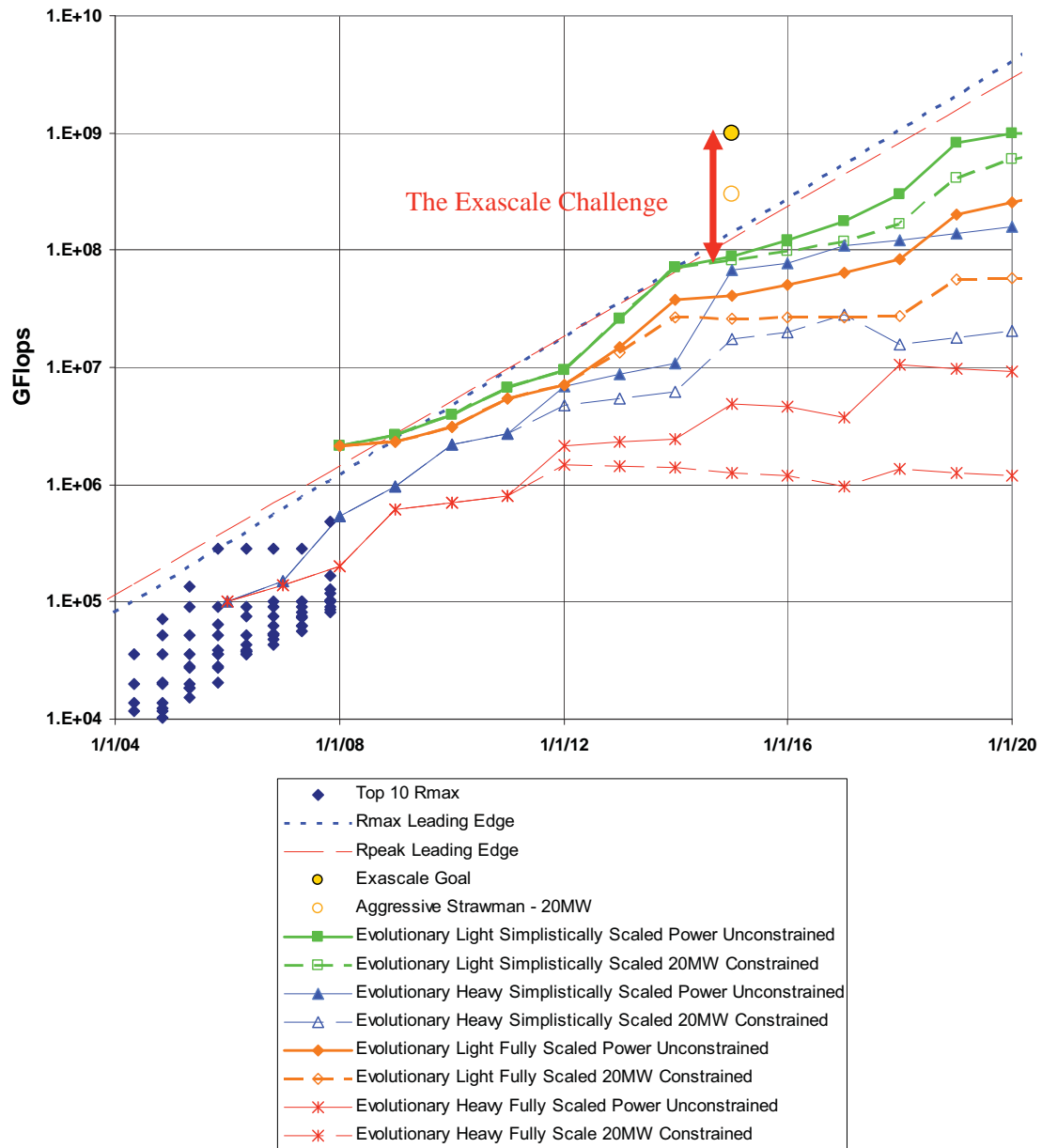


Figure 2.2: Exascale goals — Linpack.

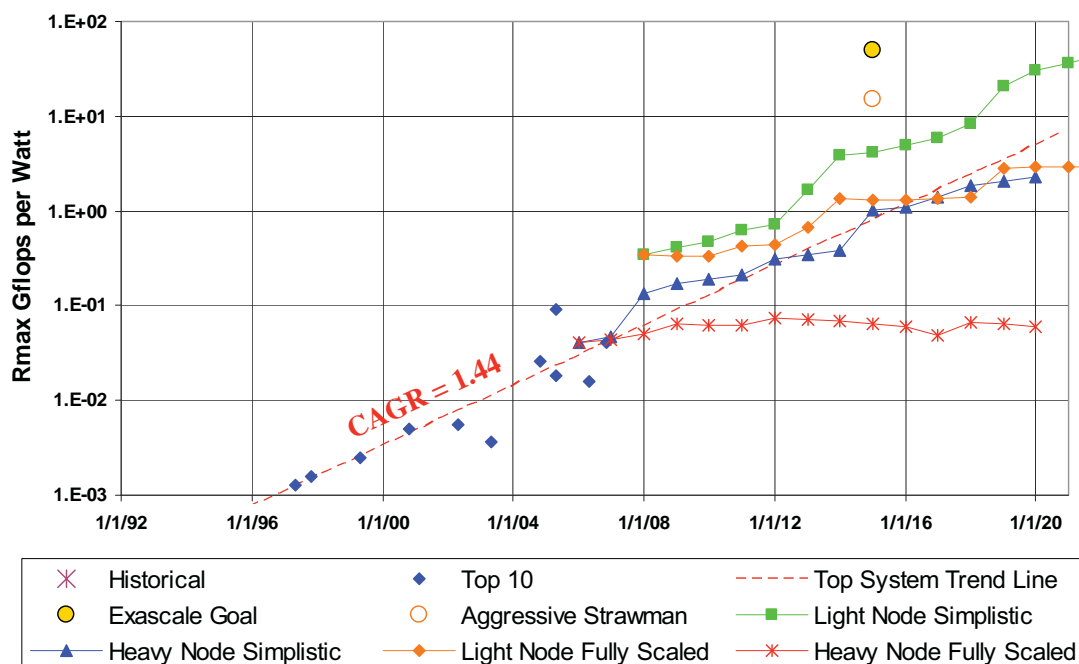


Figure 2.3: The power challenge for an Exaflops Linpack.

2.3 The Summary Extrapolations

2.3.1 Power

Figure 2.2 is a copy of Figure 8.1 from the original report, and summarizes the results of extrapolating the performance of both of the two “today” baselines from Table 2.2 through the expected evolution of technology, assuming nothing out of the ordinary happens *i.e.*, no “clean sheet” opportunities.

There are four lines associated with each of the conventional strawmen: one that assumed no maximum power constraint other than some maximum number of racks at some maximum feasible power per rack, one where the maximum power is limited to 20 MW (the original target power), one labeled “simplistically scaled” where the energy per bit moved or accessed from memory scales down with technology, and one labeled “fully scaled” where such energies do not change. The third is highly optimistic; the fourth pessimistic. We expect reality to lie somewhere in between these two curves.

There is also a point in 2015 at 1 exaflops representing the study’s goal, and a point below it representing the aggressive strawman if power was limited to 20 MW.

The results indicate the severity of the performance and power problem. None of the design points studied reached the exaflops level of performance for 20 MW. Further, only two design points reach the desired exaflops performance — the aggressive design in 2015 at a cost of 67 MW, and an optimistic (unrealistic) extrapolation of the lightweight node strawman around 2020. Figure 2.3 (a copy of Figure 8.3 from the report) presents the same extrapolations with “Gflops per watt” as the y-axis. In the 2015 timeframe everything other than the aggressive design is off by one to three orders of magnitude, and the aggressive design itself is still off by a factor of 3.

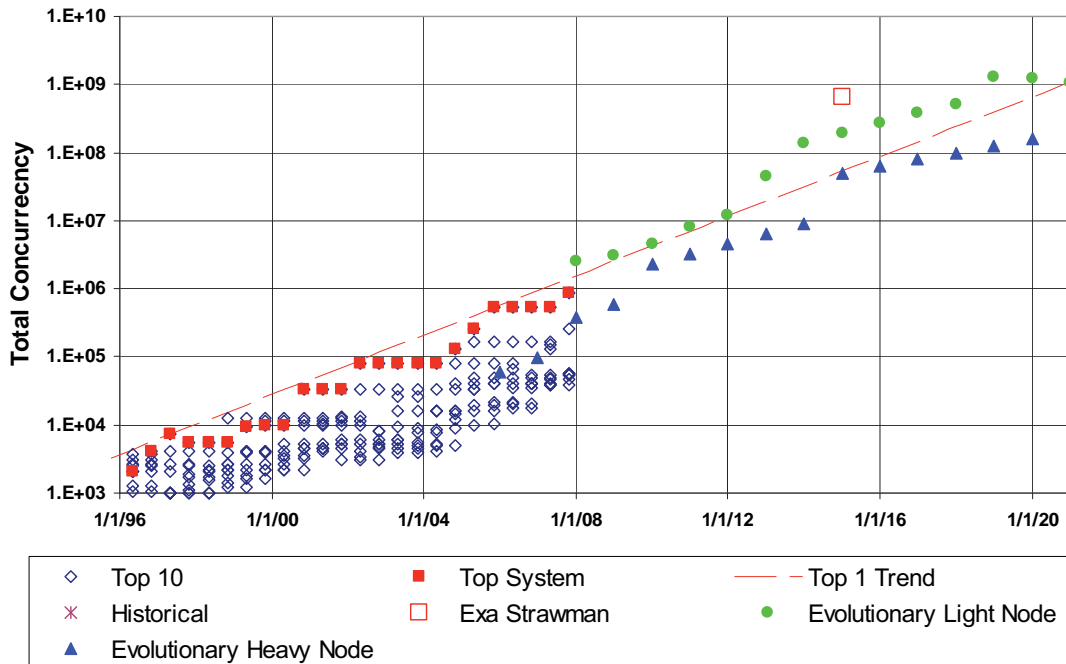


Figure 2.4: The overall concurrency challenge.

2.3.2 Concurrency

Figure 2.4 documents another relevant part of the study — an estimate of the concurrency seen in the various strawmen. Here, concurrency is defined as the number of operations (flops in this case) that must be started in each and every cycle by a program over the entire duration of an application in order to achieve exaflops performance. Again, the numbers are enormous, reaching a billion for the only two points that reach an exaflops. This is at least three orders of magnitude larger than the degree of concurrency exhibited by any machine today.

2.3.3 Memory and Bandwidth

A rule of thumb is that a supercomputer needs about a byte of memory capacity and a byte per second of memory bandwidth for each flop executed per second. Today's top machines are in the 0.1 to 0.3 range for both. As is listed in Table 2.2, the memory capacity of the aggressive design is considerably smaller, as is the bandwidth to memory as discussed in Section 2.2.1.

There are no obvious ways of fixing either gap with current technology and within any sort of acceptable power budget. Later in the report, we discuss techniques (such as “new-era” weak scaling in Chapter 4) that can tolerate this new imbalance between memory and computation.

Chapter 3

Extreme Scale Software Execution Models and Metrics

3.1 Execution Models

Before discussing the potential challenges of extreme scale computing, and potential solutions to those challenges, it is appropriate to define the environment in which programs that perform such computation will find themselves. This chapter introduces the concept of an “execution model” as the basis on which extreme computation must be specified, along with several examples. It then defines several general metrics by which extreme scale systems are likely to be measured and evaluated. Later chapters will elaborate on these metrics.

3.1.1 Models of Computation

A model of computation is a paradigm for organizing and carrying out computation across all levels of the computer system stack from programming models and languages through compilers and runtime systems to operating systems and system and micro architectures. It provides the conceptual scaffolding for deriving each of these system elements in the context of and consistent with all of the others. This paradigm suggests a decision chain to which each layer contributes that ultimately determines when, where, and how every operation of a computation is performed. An execution model is not a programming language although it may strongly influence the underlying programming model semantics of which the language is a representation. It is not a computer architecture although it establishes the needs for low level mechanisms the architectures must support and provides the governing principles that guide the structures and actions of a computer architecture in the performance of a computation. And, it is not a virtual machine isolating the abstractions above it from the implementation details below because it cross-cuts all layers from programming language to architecture influencing all aspects of the operation of all system layers in concert.

Models of computation serve as the continuum medium of existence and evolution for the sub-domain of computing systems, perhaps most predominant in high-performance computing. The evolution of high-performance computing over the previous six decades has been marked by dramatic phase changes delineating sometimes overlapping major epochs in how we reason about high performance computations. Driven by advances in underlying enabling technologies and the opportunities and challenges they present to the design and implementation of HPC systems, execution models have been changed to enable their most effective application by exploiting those opportuni-

Execution Model	Device trends	Architecture trend	System Software trend
Von Neumann	SSI devices	Scalar instrs.	Scalar compilers
Vector Parallelism	MSI devices	Vector instrs.	Vectorizing Compilers
Shared-Mem. Parallelism	VLSI microprocessors	Cache coherence	Multithreaded OS and runtime
CSP with Bulk-Sync. Parallelism	VLSI microprocessors	Interconnects	Message-passing libraries (MPI)

Table 3.1: Examples of Past Successful Execution Models

ties while simultaneously addressing their challenges. During each epoch, incremental technology advances have been incorporated with incremental modifications to architecture, operating systems, and compilers largely keeping the programming model essentially invariant along with the foundation model of computation. However, past a critical threshold, continued optimizations become untenable and a metamorphosis catalyzes the emergence and adoption of a new model of computation inaugurating new opportunities, design points, and performance as well as other operational properties.

Examples of models of computation over the history of HPC include the Aiken Harvard model, the von Neumann model, the vector model, the SIMD array model, the dataflow model, the systolic model, the communicating sequential processes model (CSP), and the multithreaded shared memory model. Not all execution models (*e.g.*, dataflow) proved commercially successful at the programmer-visible level (although models like dataflow did migrate into the CPU's microarchitecture in the form of "out of order" execution). Others, such as SIMD, have impacted in multiple forms again and again. Successful execution models have primitives that are well matched with device, architecture, and software technology trends, as illustrated in Table 3.1.

The most recent MPI epoch is remarkable in its longevity and continuity over more than two orders of magnitude in multiple dimensions of enabling technologies and metrics (flops, bytes capacity, bps data transfer). This is perhaps most readily apparent by the concerns about continued support of legacy MPI code base; a concern never previously realized for any other current parallel programming methodology. But that very duration has stretched the gap to the breaking point. The current model of computation, CSP with Bulk-Synchronous Parallelism, is no longer capable of supporting the most effective exploitation of current and future generations of implementation technologies, addressing the many new challenges they impose (such as massive concurrency with asynchrony), or facilitating the scale of computation to the ExaFlops performance regime that is required within a decade's time. Past methods of extending delivered performance are no longer tenable. Power constraints, now perhaps the most dominant challenge, are inhibiting continued growth of clock rates, once a major source of performance improvement. Multicore components have almost universally replaced single processor devices imposing an entirely new level of user-exposed parallelism while aggravating chip I/O, cache behavior, and memory bandwidth problems as well as programming methodologies. Additional departures from conventionality include new instruction set architectures, new hardware structures, and accelerators such as GPGPUs, game machines (*e.g.*, IBM Cell SPE), and attached array processors (*e.g.*, ClearSpeed). All of these breaches of CSP conventionality are symptoms of an impending phase transition in HPC. And as has always happened before in such cases, it is the change in the foundation model of computation that will fully establish the new direction for the next epoch.

Then what exactly is a model of computation if it is not the manifestations of architecture, system software, and programming models that visibly reflect it? One perspective is that an execution model is a set of governing principles that guide the form and function of computation. This includes the nature of the state that initiates, evolves, and is finalized throughout the compu-

tation to its conclusion, the atomic operations that may be performed on elements or compound structures of such elements, the semantics of flow control and concurrency of operation, the means of coordination, cooperation, and synchronization, the name spaces and their interrelationships of the data and possible the control elements as well, the innate reflection for control of hierarchy, encapsulation, modularity, means of manifesting work-flow, distribution, and asynchrony of tasks. This laundry list of pieces in ensemble constitutes a paradigm and the different models of computation employed over the decade may be distinguished by these various attributes. More generally, a model of computation answers the broad question of how we structure and name data and instructions and how we interrelate the two. Within the realm of parallel models of computation, this generality extends to how we harness, distribute, and control concurrency of action.

A subtle question that drives an HPC phase change by transitioning between models of computation is how to assess when a new model constitutes a superior paradigm with respect to a previous one being replaced? A partial answer comes from consideration of the set of underlying factors that determine efficiency of parallel execution. These are starvation, overhead, latency, and contention. Starvation is the factor requiring sufficient algorithm concurrency to drive all parallel physical elements (> 100 million cores, > 10 billion-way program parallelism) and resource allocation (dynamic load balancing) to ensure that all cores in a multicore system have useful work to do. Overhead is the factor requiring mechanisms providing critical path services to exhibit minimum response time essential for efficient exploitation of parallelism for ultra scalability. Lightweight synchronization, distributed global address translation, process instantiation and termination, thread scheduling and context switching, load balancing, and communication exemplify overhead mechanisms of importance. Latency is the efficiency factor requiring delays to critical resources due to latency of information transfer and service request to be mitigated through minimization and hiding (overlapping) such as to memory or remote computing/data resources. Contention is the factor requiring that delays to critical resources due to blocking from simultaneous service requests of shared resources be minimized and mitigated with, for example, sufficient system area network bandwidth, memory bandwidth and chip I/O, and ALU throughput with respect to demand. Within the context of these factors the effectiveness of an execution model may be assessed. But the model must also leverage the technologies it is intended to exploit. Thus it is this interplay of semantics and physics that must be coordinated.

The CSP model has been so effective because it provides this close match between the strengths of the technologies and the semantics of the model. Specifically the CSP process maps cleanly to the physical processor. The parallelism to be exposed has to match the number of processors in a system. At scales of a few hundred to a few thousand this worked well, especially for weak scaling where the data size grew proportionally with the scale of the system. The static mapping eliminated much of the overhead. Latency was mitigated by maintaining much more intra-process execution to message passing (but it also requires that there be enough memory per processor). The vector model worked because it allowed faster technologies through pipelining. The SIMD array model was best when technology density permitted many more modest processor and memory nodes to work together but when clock rates were slow enough that the instruction broadcast time was not prohibitive. Dataflow didn't work well as conceived in the 1980s in part because it was memory and communication intensive with synchronization overheads exceeding the work of the fine grain operations.

3.1.2 Desiderata for an Extreme Scale Execution Model

What then are the requirements of a model of computation that will match the future capabilities and opportunities of extreme scale systems while addressing their limitations? Considering the

critical factors mentioned earlier, several attributes can be established. Concurrency must exceed a billion-way parallelism. This suggests that the overhead for managing such parallelism has to be lower both to expose finer grain parallelism (overhead lower than the useful work per independent action). A new model of computation has to change the synchronization semantics, removing global barriers, and exposing new levels of algorithmic parallelism. It has to provide dynamic adaptive resource and task scheduling to respond to unpredictable ordering of execution such as varying latencies, contention, and priorities. Such a model has to maintain some level of global name space and manage the address translation with low overhead. The model has to permit a wide variation in implementations from architecture up to programming languages.

Most importantly, a model of computation for future parallel system implementation in the pan-exaflops performance domain must enable rational and quantifiable co-design of the separate but interrelated layers of the system stack to guide the development of the semantics and structure of each.

Extreme Scale systems are characterized by new device technologies with significant power constraints and by computer architectures that will build on manycore processors with $O(10^3)$ cores per socket. A successful Extreme Scale execution model must make explicit the key performance factors in future Extreme Scale systems which include concurrency, locality, energy, and overhead. To that end, we outline the following desiderata for an Extreme Scale execution model:

1. Asynchronous lightweight tasks and communications

- Motivation: need asynchrony to hide latency and handle variability among cores
- Challenge: scheduling with bounded resources (adapt across eager vs. lazy scheduling for starvation vs. contention modes)
- Related topics: control vs data-driven initiation/termination of tasks

2. Explicit locality model

- Motivation: locality is key to efficient parallelism
- Challenge: portable and hierarchical abstractions of locality
- Related topics: co-locating distributed tasks and distributed data c.f., Sequoia and X10 execution models

3. Scalable Coordination and Synchronization

- Mutual exclusion (transactions)
- Producer-consumer (streams)
- Collective synchronization (barriers, phasers)
- Other collective operations (reductions)
- Point-to-point synchronization (semaphores)

4. Abstract performance model

- Parallelism
- Locality
- Energy

3.2 Metrics

In this section, we summarize metrics that can be used to evaluate the effectiveness of new system software technologies for Extreme Scale systems. Though there has been significant amount of past work on metrics for application characteristics (Chapter 4) and for hardware characteristics [62], less attention has been paid to metrics for the system software that bridges the two. This is unfortunate because it is widely recognized that limitations in system software can cause algorithms that are highly scalable in theory to fall short by one or more orders of magnitude in realizing their full potential on parallel hardware. Examples of system software characteristics that contribute to this attenuation include OS scalability limitations, lack of locality/affinity management, lack of adaptation and self-awareness, compiler optimization limitations, thread/task creation overhead, synchronization overhead, and other overheads in runtimes for scheduling, memory management, communication, performance monitoring, power management, and resilience.

As we study metrics for system software, we observe that system software for Extreme Scale systems must strive to balance conflicting goals in its management of *concurrency*, *locality*, and *energy*. The conflicts among some of these goals are well understood from past research. For example, there is a natural tension between concurrency and locality when spreading computations across multiple cores can improve concurrency but degrade locality. Likewise, frequency and voltage scaling for energy optimization may degrade concurrency and time-to-completion if the frequency scaling resulted in the slowdown of a task on the critical path. What we desire is a single combined metric or figure of merit that can capture all these trade-offs. To that end, our proposed single combined metric is to build on the idea of *energy-delay product* [138] as follows.

The cost of an execution of application A on an Extreme Scale platform with system software S and hardware H can be expressed as

$$C(A, S, H) = \text{TotalEnergy}(A, S, H) \times \text{TotalElapsedTime}(A, S, H).$$

This C-A-S-H metric can be used as the starting point for a number of detailed evaluations. For example, to evaluate the cost improvement delivered by a new system software stack, S_{new} , relative to an existing system software stack, S_{old} , we can compute the ratio, $C(A, S_{old}, H)/C(A, S_{new}, H)$, for some number of (A, H) pairs. The choice of applications, A , can be driven by a small number (say 3 to 5) of grand challenge applications that are representative of workloads of importance to mission partners, and the choice of hardware platforms, H , can be simulated by analytical models such as the strawmen Exascale systems introduced in [62] or by extrapolating from current Petascale systems.

The $\text{TotalElapsedTime}(A, S, H)$ and $\text{TotalEnergy}(A, S, H)$ sub-metrics can in turn be used for more detailed point-to-point comparisons. For example, the TotalElapsedTime metric can be used to compare the scalability of two different system software stacks, as discussed in Appendix B. Also, since vertical locality (or the lack thereof) has been identified as a significant contributor to energy costs, the TotalEnergy can be used to compare the locality management capabilities of two different system software stacks. Microbenchmarks akin to HPCC can also be developed to evaluate performance per unit energy (*e.g.*, operations/Joule and bytes-transferred/Joule), which is correlated with the reciprocal of the C-A-S-H metric. As yet another example, real-time applications that work with a fixed value for TotalElapsedTime (deadline) can use the C-A-S-H metric to compare the energy efficiency of two software stacks that satisfy the same deadline.

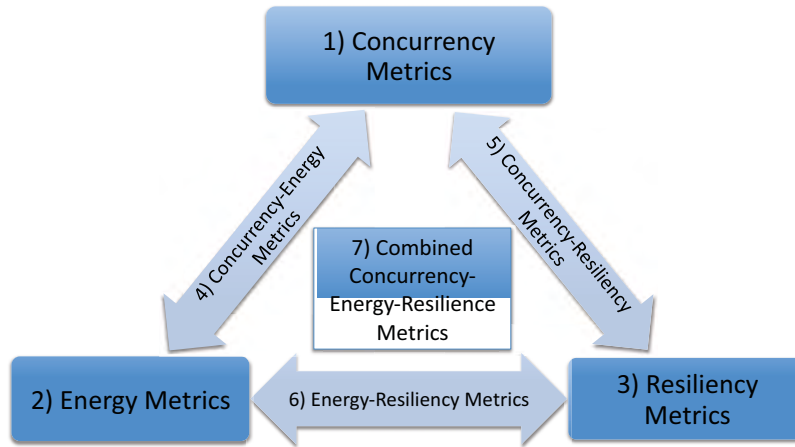


Figure 3.1: Structure of Metrics for Concurrency, Energy, and Resiliency

Chapter 4

Challenges in Developing Applications for Extreme Scale Systems

This chapter addresses the challenges facing the development of applications for Extreme Scale computing systems. These challenges lie both in understanding how much system resources are needed to support applications as their requirements change in size and complexity, and in the converse, namely how efficient applications may be as a function of how many resources are made available to them. In particular, Section 4.1 summarizes the discussions to date within the community about applications requirements and system size. Section 4.2 then discusses in more detail what “application scaling” means. The succeeding sections then consider different classes of applications. Section 4.9 concludes by summarizing application “sweet spots” in terms of resources for extreme scale as we currently understand them.

4.1 Application Overview

Application scaling addresses how applications may port to new systems in two respects: how data sets and problem sizes may grow and fit in new machines with more resources, and/or how existing applications may adapt when “bigger” systems become available for their execution. Discussions on how and why such scaling may occur have been topics of considerable debate within the community over the last few years. In particular, during 2007, there were numerous meetings held to investigate future computing applications and hardware/software requirements as they might exist at the exascale level. These meetings included:

- Three DOE Exascale Townhall Meetings [2]
 - Lawrence Berkeley National Laboratory (LBL) (April 2007) [3]
 - Oak Ridge National Laboratory (ORNL) (May 2007) [4]
 - Argonne National Laboratory (ANL) (May/June 2007) [5]
- Council on Competitiveness meeting on Exascale Applications [87]
- Frontiers of Extreme Computing 2007/Zettaflops workshop (October 2007) [6]

At the DOE-sponsored meetings, there were extensive discussions of those applications that could and should scale to exascale. At the Council on Competitiveness meeting, commercial HPC users discussed applications that they believed could benefit from extreme scaling.

Exascale Software Study

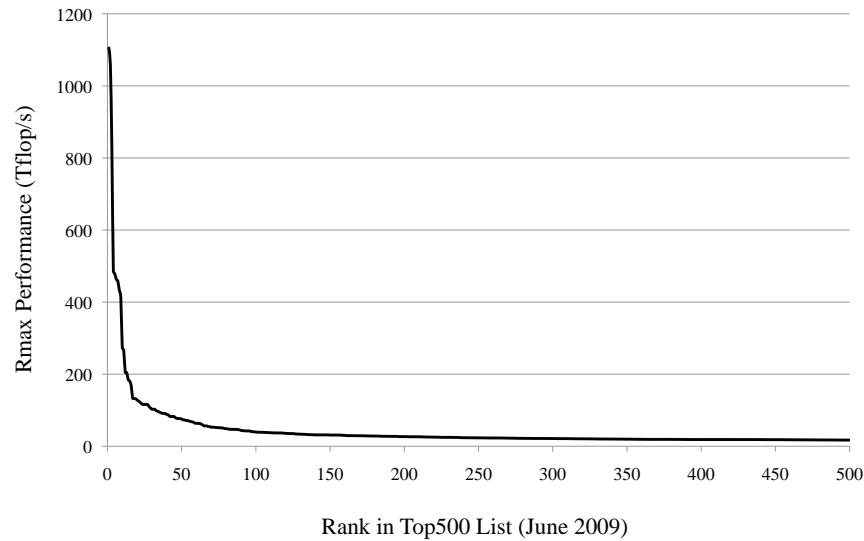


Figure 4.1: Rank-ordered distribution of Rmax in Tflop/s for systems in the June 2009 Top500 list [1]. 2 of 500 systems have Rmax > 1000 Tflop/s (1 Pflop/s). 468 of 500 systems have Rmax < 100 TFlop/s. 421 of 500 systems have Rmax < 50 TFlop/s.

Not all existing applications will scale to terascale, petascale, or on to exascale given current application/architecture characteristics. The reasons for limited scaling are potentially many, including (but not limited to):

- Parallelism
 - Terascale — $O(10^5)$ threads
 - Petascale — $O(10^8)$ threads
 - Exascale — $O(10^{11})$ threads
- Locality
 - Vertical — temporal locality (*i.e.*, reuse). One exploits vertical locality by moving data up and down a node's memory hierarchy.
 - Horizontal — occurs as a result of node-level data decomposition. One exploits horizontal locality through domain decomposition — by putting a portion of the application data on each node of the machine.
- Bottlenecks
 - Memory bandwidth
 - Bisection bandwidth
 - I/O bandwidth

Of those applications that operate at sustained terascale today, only a small fraction of those applications will be successful at reaching petascale. Hopefully, lessons learned when moving applications to petascale, will permit a reasonable fraction of petascale applications to scale out to exascale levels.

Insight into the current state of application scaling can be found by looking at the June 2009 Top500 list [1]. Figure 4.1 shows the rank-ordered distribution of the Rmax peak performance in Tflop/s for the systems in this list. While the Los Alamos National Laboratory's Roadrunner and Oak Ridge National Laboratory's Jaguar systems have broken the petascale performance barrier on high-performance LINPACK, more than 93% of the top 500 systems have Rmax below 100 Tflop/s. Clearly there are not substantial numbers of HPC applications currently running at near-petascale levels, which underscores the challenge for application enablement at the exascale level. As we examine future exascale application footprints, we will also need to analyze existing applications and develop models for application scaling to develop projections for applications running at petascale and exascale. Eventually, these models will be validated by petascale application development work being performed throughout DOE to provide applications for the LANL RoadRunner [7], and work being performed by the University of Illinois Urbana-Champaign (UIUC) to study scaling as they prepare to receive the National Science Foundation (NSF) Tier 1 Blue Waters sustained Pflop/s system [8].

4.2 Application Scaling

Application scaling remains a major challenge in the utilization of high-end parallelism. Of applications that operate at sustained Terascale performance today, only a small fraction is expected to be successful at reaching Petascale and an even smaller fraction at Exascale. Further, even for applications that reach Petascale performance today, the nature of scaling necessary to obtain Petascale performance on an Extreme Scale departmental system will raise new challenges for rewriting the application to address the concurrency and locality requirements of such systems. In this paper, we argue that the existing software "stack" is a major contributor to these scalability limitations, and that the approaches discussed in the following sections could have a significant impact in removing obstacles to scaling.

There are three primary ways to scale applications:

- Strong scaling — apply more resources to the same problem size to get faster results.
- Weak scaling — apply more resources to larger problem sizes to do more within a tractable time period.
- Temporal scaling — run an application longer.

We will examine the impacts of strong and weak scaling below, and defer more formal definitions to Appendix B. Temporal scaling does require additional resources, but those are spread over time. Temporal scaling does not provide additional work within a timestep. Thus there is no increase in the flop/s rate.

4.2.1 Strong Scaling

Strong scaling refers to the concept of applying more resources to the same problem size to get results faster. Unfortunately, few applications are amenable to strong scaling, so we cannot rely on strong scaling to move applications from petascale to exascale. As an application is strongly

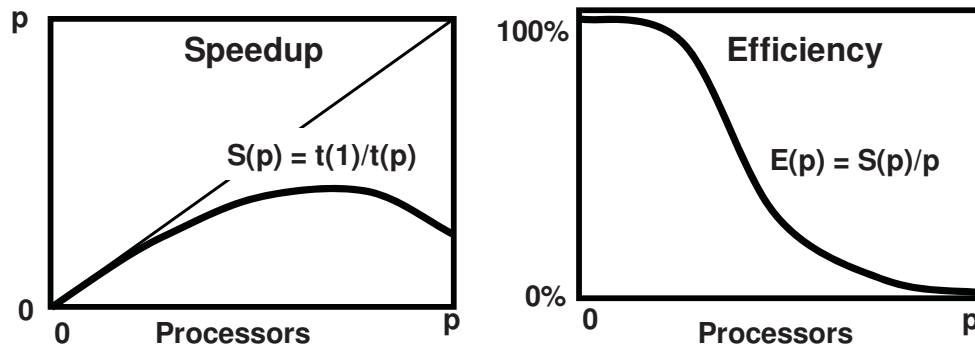


Figure 4.2: Conceptual speedup and efficiency curves for strong scaling

scaled, the work at a node/processor/core decreases and the relative overhead increases. Speedup may initially equal the number of processors, but eventually the amount of overhead causes the slope of the speedup curve to flatten. At this point, adding processors does not cause the application to run faster. Eventually, it is possible that overhead grows so rapidly that adding processors actually causes the time to solution to flatten or increase and speedup to flatten or decrease. An application that demonstrated reasonable scaling over three orders of magnitude increase in the number of processors is the first principles molecular dynamics “Qbox” code that won the 2006 ACM Gordon Bell Prize for “peak performance” with over 200 Tflop/s sustained performance (56% efficiency) on the LLNL BlueGene/L [9]. More recent winners of the 2008 ACM Gordon Bell Prize further underscored the importance of algorithmic innovations that attain very high levels of spatial and temporal locality.

4.2.2 Weak Scaling

Weak scaling refers to the concept of adding work as an application is run on more processors. By adding work, it is possible to ensure that overhead does not destroy performance. Traditionally, weak scaling has been accomplished by adding work due to spatial scaling. However, there are additional sources of scaling — discussed below and referred to in this report as “new-era” weak scaling — arising from new application trends in which additional work is done per datum *e.g.*, multi-scale, multi-physics, interaction analysis, and data mining. Weak scaling permits the user to look at larger or more complicated problems and use the additional processors to solve larger problems, obtain better resolution, or learn more about the phenomenon being examined.

Traditional weak scaling occurs in classical mechanics simulations, where either (1) larger problems are examined or (2) the grid size and time-step interval are reduced. Solving larger problems (*e.g.*, modeling the airflow around an entire airplane versus modeling the airflow over a section of the wing) results in a situation when memory scales nearly proportionally with work. In contrast, when the grid size is reduced (refined) in a 3-D mechanics simulation, the time step also needs to be reduced thereby increasing the amount of work relative to the amount of memory. When scaling in three dimensions, a $3/4$ power rule applies to the amount of memory required whereas a $2/3$ power rule applies when scaling in only two dimensions. Thus, for these applications, the required increase in memory size for a $1000\times$ increase in work is $180\times$ and $100\times$ for 3-D and 2-D applications respectively.

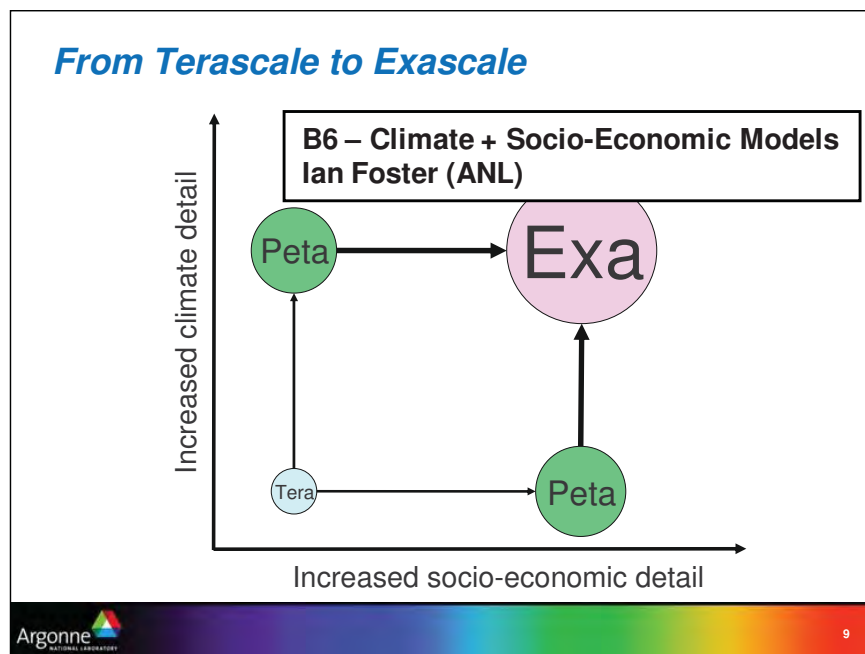


Figure 4.3: New-era weak scaling example

4.2.3 New Era Weak Scaling

“New-era” weak scaling typically adds extra work through one or more of the following:

- Multi-scale
- Multi-physics (multi-models)
- New models
- Interactions
- Mitigation analysis
- Data mining
- Data-derived models

This list is a product of aggregating materials presented at the DOE Exascale Townhall meetings. Figure 4.3 presents a slide indicating a conceptual scaling of the combination of climate and socio-economic models. In this figure, the climate and socio-economic models would be scaled out with increased detail then the analysis of the interactions would further boost the scale. Compared to traditional weak scaling, it can be more difficult to predict application footprints for new-era weak scaling.

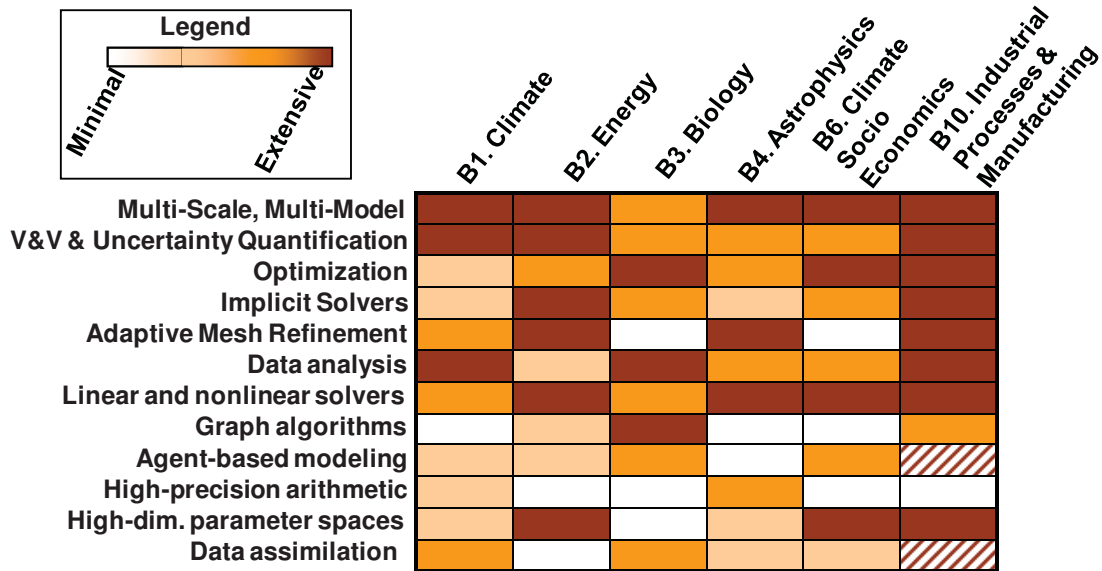


Figure 4.4: Application characteristics from Group B7 Mathematics and Algorithms, David Bailey, et.al.

4.2.4 Exascale Application Scaling

Nothing implies that we should expect a single silver bullet to enable applications to scale to exascale. We cannot expect extensive strong scaling in applications as the parallel computing “laws” of work versus overhead will still hold. “Ensemble” calculations cannot turn capability problems into capacity problems. Temporal scaling will not provide greater instantaneous amounts of “work”. Anticipated new-era weak scaling may drive demanding footprints requiring more memory than the 180x implied by the $\frac{3}{4}$ power law. In addition to the need for extensive memory, there may be reduced data locality in the data-mining/data-intensive portions of the applications that examine domain interactions. These application characteristics may be far from the locality, reuse, and regular communications of high-performance LINPACK. On the other hand, anticipated new-era weak scaling may reduce memory demand when $O(n)$ scratch data is used with $O(n^m)$ data-mining work with m integrated modules. Some hardware characteristics may provide relief for other hardware deficiencies *e.g.*, it may be possible to trade memory footprint for disk bandwidth for application checkpointing. As stated above, exascale applications may benefit from current petascale application research to increase the amount of parallelism.

At the DOE Exascale Townhall meetings, David Bailey (LBL) presented a table of application characteristics for six application classes. This figure is represented in figure 4.4. These application classes were different “tracts” at the meetings. In this figure, the intensity of the block shading identifies the anticipated impact of the listed application characteristics. In two cases, this figure was modified to add information obtained from an Exascale Application discussion session run by the Council on Competitiveness. In those two cases the hatched blocks show that these application areas were modified from minimal impact to extensive impact.

4.3 Emerging Extreme Scale Applications

Candidate applications for extreme scale include:

- traditional HPC applications scaled up from petascale,
- traditional or new applications that are to run at petascale on departmental class systems,
- coupling together of multiple petascale applications to form an exascale application,
- emerging data mining applications, and
- real-time departmental applications.

Within these categories we wish to consider the following kinds of scaling:

- **Strong scaling:** Those that could solve the same problem as today but $1000\times$ faster. For example, this category may include real-time CFD to stabilize a physically morphable airplane wing (effectively recalculating an AVUS calculation that now takes hours every few seconds).
- **Weak scaling:** Those that would solve the same problem as today but at a $1000\times$ larger scale. This is normally called “weak scaling”. In this category for example might be global weather models at sub 1km resolution (effectively solving a WRF calculation at the granularity used today to forecast hurricanes but at the scale of the entire earth’s atmosphere).
- **Increased time steps:** Those that would solve the same problem as today but with $1000\times$ more time steps. In this category for example might be local weather models at climatic timescales (effectively solving a WRF calculation at the size today used for medium-term weather forecasting but projecting out for centuries).
- **Increased resolution:** Those that would solve the same problem as today but at $1000\times$ more resolution (or increased physics and chemistry in every cell). In this category might be for example global ocean and tide models that include micro-features such as wave refraction and coastal turbulent mixing. This may include coupling.
- **New approaches:** Those that solve entirely new problems than those solved today. These may include emerging applications in biology and social networks data mining.

Clearly Exascale applications could have aspects of any combination of the above; for example one could perform a calculation at $100\times$ resolution and $10\times$ more timesteps, for a total of $1000\times$ more computation than is possible today. In the following sections we identify some specific exemplars of each and draw some general conclusions.

4.4 “Traditional” HPC Applications at Exascale

A good example of a scale-up of a traditional HPC application is next-generation hurricane modeling and prediction that requires ultra-high-resolution of gradients across the eyewall boundaries (at 1 km or less), and representation of the turbulent mixing process correctly (at 10 m or less). To do this over a local region, as for example a hurricane eye-wall, requires a petascale calculation. The requirements are for 100 kilometer square outer-most domain at 10 meter horizontal grid spacing and 150 vertical levels, 15-billion cell inner-most 10 meter nested domain, with a model time step

of 60 milliseconds. The calculation takes (at 100,000 tasks) 100 MB per task of data not counting buffers, executable size, OS overhead, etc. A petascale run generates $24 * 1.8$ terabyte datasets = 43.2 terabytes per simulation day assuming hourly output of 30 three-dimensional fields. Assuming an integration rate of 18 machine hours per simulated day at a sustained petaflop, the average sustained output bandwidth required is about 700 MB/second.

Now if one were to envision exascale extensions, scaling climate time frames is a “capacity” problem. A $1000\times$ faster machine with no more main memory would allow one to model a very limited local region at the rate of 3 simulated years per day. However, an entire hemisphere of earth at “hurricane eyewall” resolution that might for example capture “butterfly effects” is a capability problem requiring a sustained exaflop and 10,000 Gbytes = 10 petabytes of main memory to model the earth at 10m resolution at the rate of 1 simulated day per day.

4.5 Coupled Models

In addition to traditional HPC applications, there is an opportunity to use an exascale facility to enable adding additional information to a model, for example adding ecological information, such as forest growth, to models of weather and climate. A grand challenge in geoscience is the addition of clouds to ecological modeling, especially since clouds and cloud-formation processes interact (in both directions) with the ecosystem.

Ecological models attached to a high-resolution model of climate change that already have petascale applicability may provide a good starting point. Ecologists have several embarrassingly parallel applications that can be “easily” scaled to a petascale system, including:

- stochastic processes¹ that need replication
- parameter sensitivity analysis
- heuristic optimization

Such problems are extremely common in ecological applications, as we elaborate here.

1. Stochasticity: Simulation-based ecological models often incorporate demographic stochasticity (random birth/death/movement, etc), environmental stochasticity (random components of climate forcing, resource availability, etc), and/or genetic stochasticity (random mating, mutation, etc). Outcomes are thus stochastic as well, and ecologists wish to ask questions like, “What is the simulated probability that the population size will fall below X within 100 years?” The simulation model must therefore be independently repeated (usually 100s-1000s of times) to generate a distribution of outcomes.
2. Parameter sensitivity (or more generally, model sensitivity): The “true” parameters of ecological models are rarely known, and in fact there are often disagreements about the form of the equations governing those processes. Consequently, ecologists frequently want to characterize the sensitivity of outcomes to input parameter values and model assumptions. This also requires repeated simulation.
3. Optimization: There are (at least) two distinct types of optimization questions that ecologists commonly ask. The first involves fitting parameters to observed data. In all but the most trivial models, it is impossible to use analytical or even simple approximating techniques to

¹A stochastic process is one that has both predictable and random components in its formal description.

identify maximum likelihood estimates of parameters. Increasingly, ecologists are turning to stochastic optimization techniques such as simulated annealing, or the use of various implementations of Markov Chain Monte Carlo to simulate posterior probability distributions in a Bayesian framework. Secondly, applied ecological models often implement heuristic optimization algorithms as decision tools (*e.g.*, identifying the optimal spatial configuration of a land reserve system, given some cost criterion). As with parameter estimation, the simpler algorithms used in the past have been shown to be deficient in complex settings, but more reliable methods require many repeated simulations and long run-times. There is a tremendous need for HPC solutions that can deliver results sufficiently quickly even for models involving many parameters, fine-scale spatial and temporal resolution, and stochastic processes.

Putting this all together, it is clear that the compute time can be overwhelming when coupling one or more of the above procedures with even a moderately complex ecological simulation model. Specifically, some model examples include predicting evolution of a collection of interacting species, spatial spread of a disease, or the dynamics of a specific ecosystem. Taking the last example, imagine a regional-scale ecosystem model, the core of which is deployed as a small-scale HPC application (*e.g.*, a single simulation that takes days to complete on a cluster with dozens of nodes). Indeed, the ATLSS group² based at UT/ORNL has spent ~ 10 years developing and refining a model that integrates a variety of complex and interacting sub-models to simulate key Geoscience and environmental components of the Florida Everglades; one sub-model has already been parallelized to run on 60+ nodes. Even if a researcher demands just several hundred stochastic replications in such a simulation, performed for each of 100 possible configurations of a proposed reserve system, there would be significant benefit from hierarchically organized parallelization, to enable a 100k-processor system run (imagine a multi-hundred simultaneous, distributed instantiation of the ecosystem simulation, which itself might be a 64-node data-parallel application). Whether the envisioned exascale system even provides the right architecture for this application could be debated, but the point is that it does not require significant effort to scale up moderately sized ecological models to result in large computational needs, resulting in the ability to address relevant and interesting problems.

Data integration would be critical to success. A candidate calculation would involve evolutionary correlations of networks and functions (phenotypes). To the extent that ecologists are able to refine mechanistic mathematical models in a way that is increasingly faithful to reality, one could easily conceive of petascale computing demands for simulating an entire ecosystem from its underlying biological and physical components. However, it is worth pointing out that the tradition in ecology is to simplify and scale back models — indeed, to err on the side of oversimplification; “realistic” models have long been mistrusted in favor of either highly abstract mechanistic (theoretical) models and/or simple phenomenological (statistical) models. In part this is for good reason: ecologists do not yet fully rely on their own more detailed mechanistic models (there being a lack of the ecological equivalents of physical laws, testing approximating models via experimentation and observation is difficult, and each real system seems to have its own unique features). This could in fact partly be a historical artifact: few ecologists are even aware of the computational possibilities now afforded by HPC systems. In a sense, one might argue that developments in this area are limited due to apparent belief in computational obstacles that no longer exist. Opportunities for making forays into developing complex ecological simulations and to enhance model output with observed data has the potential to lead to refinement and progress in this area.

²<http://www.atlss.org>

Class	Application	Benefits from	References
Data Mining	<i>De novo</i> genome assembly from high-throughput sequencers	Large shared memory	[49, 63, 79, 150]
	Clustering and correlation analysis of galaxies from cosmological simulations	Large shared memory	[40, 58, 97]
	Interaction network analysis	Fast I/O	[40]

Table 4.1: Example data-intensive HPC applications

4.6 Exascale Data Intensive and Data Mining Applications

In talking about exascale we should also consider improving the performance of data-intensive applications, not just floating-point intensive applications. By talking to users, examining their applications, and participating in community application studies [66, 129, 130], we have identified data-intensive HPC applications spanning a broad range of science and engineering disciplines that will benefit from fast I/O and large, fast shared memory packed onto a modest number of nodes, most notably data mining and predictive science applications that analyze large model data.

In a typical *data mining application*, one may start with a large amount of raw data on disk [134]. In the initial phase of analysis, these raw data are read into memory and indexed; the resulting database is then written back to disk. In subsequent steps, the indexed data are further analyzed based upon queries, and the database will also need to be reorganized and re-indexed from time to time.

As a general rule, data miners are less concerned about raw performance and place higher value on productivity, as measured by ease of programming and time to solution. Moreover, some data-mining applications have complex data structures that make parallelization difficult [149]. Taken together, this means that (for example) a large shared memory architecture and matching shared-memory programming model will be more attractive and productive than a message-passing approach for the emerging community of data miners. I/O speed is also important for accessing data sets so large that they do not fit entirely into memory.

A typical *predictive science* application may start from (perhaps modest) amounts of input data representing initial conditions but then generate large intermediate results that may be further analyzed in memory, or the intermediate data may simply be written to disk for later data-intensive post-processing. The former approach benefits from large memory; the latter needs fast I/O to disk. Predictive scientists also face challenges in scaling their applications due to the increasing parallelism required for petascale and beyond [149]; they benefit from large memory per processor as this mitigates the scaling difficulties, allowing them to solve their problems with fewer processors.

4.6.1 Data-Intensive Balance and the Latency Gap

The growth-rate of data is exponential in many application domains. For example, gene sequence data is increasing at a rate almost faster than it can be stored, much less analyzed [40], and the same is true of astronomical data [51]. As we forecast the characteristics of data-intensive applications in the future, we find that today's supercomputers are, for the most part, not particularly well-balanced for their needs. Creating a balanced data-intensive system requires acknowledging and addressing an architectural shortcoming of today's HPC systems.

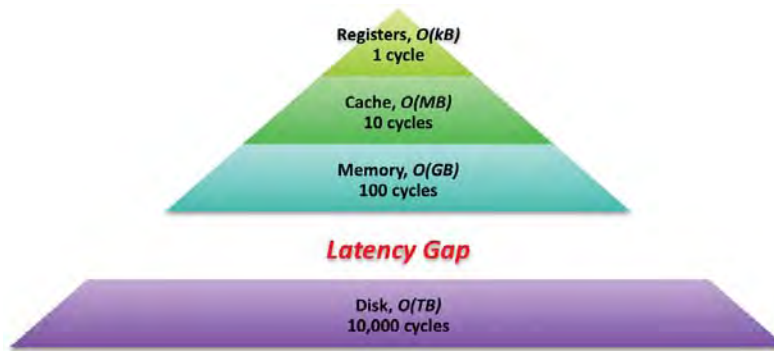


Figure 4.5: Typical memory hierarchy for today's HPC systems. Each level shows the order of capacity and typical access latency. Note the two-orders-of-magnitude gap between main memory and disk, which is expected to be bridged by new storage technologies.

4.6.1.1 Tipping the Balance to Data

When the amount of data that an application must analyze exceeds the capacity of main memory, the application may have to use many processors to obtain the amount of memory required. However, unless the calculation is highly parallel, it will incur substantial communications overhead. Also, reserving a lot of processors just for their memory is not very cost or energy effective.

Thus, a machine designed for data-intensive computing should have a large shared memory accessible to a much more modest number of processors than is common in today's systems. With a few exceptions, supercomputers in the current HPC portfolio have only 1 or 2 GB of main memory per core, while each core is capable of ≈ 10 Gflop/s. This means data-intensive calculations such as those in Table 4.1, when run on these machines, are limited, with respect to performance and capability, by the available memory rather than by the available flops. Within the nodes this translates to a need for the highest possible memory capacity and bandwidth and, for data sets that exceed the capacity of main memory, the fastest possible I/O to the next level of storage.

4.6.1.2 Hardware to Bridge the Latency Gap

While each level of the memory hierarchy in today's typical HPC systems increases in capacity, the costs of each increase are latencies that increase and bandwidths that decrease by orders of magnitude at each level (Figure 4.5). However, today's systems have a *latency gap* beyond main memory — the time to access disk is orders of magnitude greater than the access time to local DRAM memory. It is almost as though today's machines have a missing level of memory hierarchy that should read and write an order of magnitude faster than disk, a gap that is being filled by new storage technologies such as flash memory and phase change memory (PCM).

Since some data sets are becoming so large they may exceed the combined DRAM of even large parallel supercomputers, a data-intensive computer should, if possible, have a level of memory hierarchy between DRAM and spinning disk. By filling this missing level, a data-intensive exascale architecture could reap two possible benefits: to make main memory bigger or to make disk faster.

4.6.1.3 Software to Bridge the Latency Gap

Data-intensive calculations that arise outside of traditional HPC, such as data-mining applications, may not have even been parallelized yet [129], and achieving efficient data decomposition for them may be hard or impossible. For example, the fact that any gene may match any other gene or the fact that any node on the Internet may talk to any other makes it hard to partition the analysis involved in genomics or social networking, as in classical domain decomposition. As a result, these applications usually need a global address space. While technologies exist to extend the address space beyond physical memory (*e.g.*, PGAS languages), performance suffers when accessing memory outside of the current node. A data-intensive system designed to share memory across nodes should therefore have software that helps to alleviate the latency of inter-node memory accesses and, ideally, should help applications avoid such accesses by improving their data locality automatically.

Several consensus points emerged in the above studies as to the current and projected needs of data-intensive computing going towards exascale:

- Serial and/or modestly parallel data-intensive application developers expect their memory requirements to increase significantly. The reason is simple: data volume is exploding while methods to mine these data are not becoming massively parallel at the same rate; in other words, generating data is relatively easy, while inventing new parallel algorithms is hard. Therefore, shared-memory requirements of > 1 TB are expected to be the norm in many serial and modestly parallel data-intensive domains circa 2017 [130].
- Many data-intensive applications in genomics [129] and other domains have significant I/O requirements and are serial or only modestly parallel [129]. A typical data-intensive calculation today may scale poorly beyond only 128 or 256 processors [129]. This situation is not expected to improve dramatically with current trends.
- Additionally, data-intensive I/O requirements do not involve just reading or writing large chunks of data; they may also have significant amounts of small random I/O access. Therefore, simply combining disks in large RAID arrays to increase I/O bandwidth is not a panacea. Only a technology shift that reduces latency (increases random I/O access speed) can improve overall performance when there is significant random I/O.
- We estimate that, taken as a whole, data-intensive applications that need large amounts of memory, that exhibit poor scalability beyond 256 processors, or that are I/O-bound make up about 10% of the current HPC workloads. This low percentage may, in part, be due to self-elimination – there are science domains that cannot use the big flops machines effectively [129, 130]. In the future, particularly if machines that meet these needs are deployed, we expect this percentage to grow significantly.

4.6.2 Sample Data Intensive Applications

We now discuss some sample data-intensive applications, and consider their future memory and data growth.

4.6.2.1 Analyzing Interaction Networks

Interaction networks, or graphs, occur in many disciplines. These describe the relationships among objects in terms of how they are linked to each other. Often these graphs are constructed in terms of social networks [40]. Such networks (or graphs) have applicability in, for example, epidemiology

[110,122], phylogenetics [89], systems biology [44] and population biology [122] as well as in studies that combine information from these fields [40].

All these areas share a common theme — they combine information from multiple databases in order to answer questions about how the interactions lead to phenomena such as spread of disease. Key parameters that define their computational requirements include the latency to do a database lookup, and the total amount of fast storage available to the databases.

As an example, recent investigations combine social network databases with medical records and genomic profiles to explore questions such as the existence of genetic resistance to AIDS [48]. The analysis may proceed by identifying associates of diagnosed AIDS patients and analyzing their phenotype (*i.e.*, whether or not they have the disease) for correlation with genotypes such as the HLA genotype.

4.6.2.2 De novo genome assembly

Genome sequences are produced in a process known as fragment assembly: millions of tiny fragments of a genome are generated, read, and pieced together computationally. This is analogous to shredding several copies of a book, reading each fragment, and reconstructing the book.

New sequencing machines generate fragments (called reads) in orders of magnitude less time and cost than first used to sequence the human genome. Codes such as EULER-SR [49], Velvet [150], and Edena [79] automate the assembly. Each of these codes scans a file containing all reads, constructs a graph in memory that encodes all information from the reads, and then modifies the graph to produce the final genome sequence.

Key parameters that define the computational requirements for assembly are the length of the genome and the coverage, the latter being the average number of times a nucleotide is covered by a read. Interesting problems require large amounts of memory. Because of the complicated graph, none of the codes have been parallelized, which makes large shared memory very attractive.

A recent run of Velvet to assemble a plant genome with 120 Mbp (million base pairs) required 90 GB of memory [63]. Assembling the entire 3-Gbp human genome would require roughly 25x as much memory for the same coverage.

4.6.2.3 Data mining applications

Data mining is the process of analyzing large amounts of raw data and extracting useful information. The role of data mining in science and engineering will grow as the amount of data produced by experiments and observations continues to increase [134]. We anticipate applications across a continuum from simple database queries, in which samples of data are extracted for inspection, to sophisticated computations such as cluster analyses, which could run for many days. Although data need not be stored in a database, we expect the amount of database usage to increase rapidly, because of the need to efficiently organize the vast amounts of data [137]. Scientific databases in astronomy and Earth science already are terabytes in size and continue to grow. For example, the Sloan Digital Sky Survey has a 6-TB catalog archive [137], and the GEON LiDAR spatial data and indices also total about 6 TB [83]. These databases are currently stored on disk arrays and access is limited by disk read rates [137].

Future data mining investigations will involve the analysis and comparison of data from diverse sources. Metcalfe's law [68] predicts that the possibility of new discoveries grows quadratically with the number of federated databases [71]. For example, one could imagine studying the health of a community using data such as population demographics, input from various environmental sensors, and social science observations such as crime statistics. Another example is using the

ever-increasing myriad of biological data available to assist in the process of drug design [59]. By combining the information in databases on topics such as genome sequences, protein structure and function, and the biomedical literature, a researcher can generate a test hypothesis more easily because they have all the information relevant to their problem readily at hand.

Numerous data mining applications will also use large shared memory, not just fast file access. For instance, in astrophysics, many projects hinge upon assigning importance to over-dense regions in a set of points. Examples include identifying collapsed halos in a cosmological simulation, determining whether two galaxies have merged, finding clusters of galaxies in a survey, or locating dwarf galaxies in star counts. HOP [40] is a density-based clustering method, with poor scalability, that determines the location of such regions and mines the output of cosmological simulation codes such as Enzo [98]. For maximum performance such a calculation requires rapid access to the many-TB-sized simulation output files. The best approach is to park the output files in main memory.

4.6.2.4 Predictive science applications that are data-intensive

The geosciences include many applications that solve *inverse problems*, *i.e.*, problems in which measured data are supplemented with computer models to reconstruct 3D fields (often time-dependent as well) over a domain of interest.

Oceanographers in the ECCO consortium (Estimating the Circulation and Climate of the Oceans) are generating databases of ocean state as functions of space and time throughout the world's oceans. Such *ocean state estimation* supplements sparsely measured state data with a sophisticated ocean general circulation model (GCM), embodied in MITgcm. This code solves a nonlinear minimization problem by iteratively sweeping through the forward and adjoint GCM equations. These computations generate a large amount of intermediate data during the forward sweep that need to be reused during the backward, adjoint sweep.

A frequent problem for weather and climate modelers is to obtain initial conditions for subsequent simulations using atmospheric GCMs. This *data assimilation* problem is analogous to the ocean state estimation problem and is amenable to similar adjoint solution approaches [78, 146], which again benefit from the availability of additional memory.

Geophysicists within the Southern California Earthquake Center are using *full 3D seismic tomography* [146] to obtain a 3D elastic structure model of the Earth's crust under Southern California. Another class of predictive science applications exhibits only modest scalability. Well known examples are quantum chemistry packages, such as Gaussian [10] and GAMESS [11], and structural engineering packages, such as ABAQUS [12]. The equations solved by these packages, such as the matrix diagonalization step in the Hartree-Fock method from quantum chemistry, exhibit fundamental obstacles to efficient parallelization. Adding more and more processors simply does not improve performance.

4.7 Real-time Departmental Extreme Scale Applications

Extreme scale computers provide the opportunity to read in and react to an enormous amount of data in realtime. As an exemplar, consider the ability to analyze in real-time 1 million sources, such as field sensors that can produce > 1 GB/s of visual data each (*e.g.*, the sources could be hi-res. cameras with 1920x1080 64 bit pixel resolution). A machine would have to be able to read in at the rate of 1 petabyte per second assuming there was no further computation involved. Thus the ability to do petascale computing at the departmental level could enable surveillance in real time to embedded sensors on a wide scale. A detailed discussion of real-time and other specialized requirements in embedded software can be found in Appendix A.1.

September 14, 2009

Page 29

ECSS Report

4.8 Framework Technology

Enabling scientific applications on extreme-scale HPC systems is a multi-disciplinary, multi-institutional, multi-national efforts. Application code complexity is growing to a point that it is increasingly difficult to make forward progress without high-level organizing constructs. Advanced parallel languages are necessary to make programming of complex systems tractable, but they are not sufficient. Frameworks provide higher-level organizing constructs for teams of programmers that clearly segregate the roles and responsibilities of application team members along the lines of their expertise as shown in Figure 4.6. Languages must work together with frameworks for a complete solution.

Frameworks confer the following benefits:

- Separate roles and responsibilities of expert programmers from that of the domain experts/-scientist/users (productivity layer vs. performance layer).
- Define a social contract between the expert programmers and the domain scientists.
- Enforce and facilitate software engineering discipline.
- Encapsulate complex parallel implementation details in the framework so that they can be hidden from scientists and end-users.
- Allow scientists/users to code nominally serial plug-ins that are invoked by a parallel schedule.
- Support modular composition of multi-physics applications using components supplied from different developers and vendors
- Restrict code rewrites to the driver level as the hardware industry moves towards multi-core architectures with massive parallelism.
- Reduce software development costs.

The definitions and some examples are taken from the High Performance Computing (HPC) Application Software Consortium (ASC) white paper on frameworks from March 17, 2008 [72]. The state of evolution of application software can be measured in terms of level of component interoperability, given in Figure 4.7. The level of interoperability is defined by the degree to which components must conform to a set of rules set by the framework in order to achieve interoperability. We refer to three levels of interoperability:

- *Minimal Component Interoperability:* A majority of the existing commercial solvers based on legacy codes can be described as having minimal component interoperability. Individual physics models are handled by separate solvers. Therefore, the physics domains are completely un-coupled; static analysis might be performed across physics domains using file translators or common interchange file formats, also referred to as workflow coupling. The only requirement that the framework imposes on the individual solvers is that they share the same file format.
- *Shallow Component Interoperability:* Several of the leading simulation software vendors have released simulation suites that are beginning to exhibit shallow component interoperability. At this level, physics models are loosely coupled at some time step or discrete event. Each solver maintains its own internal state representation of its respective domain. Common data is exchanged using wrappers to some interchange interface over a network service. Therefore, the development guidelines imposed by the framework stop at the interface to the component.

Exascale Software Study

Developer Roles	Domain Expertise	CS/Coding Expertise	Hardware Expertise
Application: Assemble solver modules to solve science problems. (eg. combine hydro +GR+elliptic solver w/MPI driver for Neutron Star simulation)	Expert	Intermediate	Novice
Solver: Write solver modules to implement algorithms. Solvers use driver layer to implement "idiom for parallelism". (e.g. an elliptic solver or hydrodynamics solver)	Intermediate	Expert	Intermediate
Driver: Write low-level data allocation/ placement, communication and scheduling to implement "idiom for parallelism" for a given "dwarf". (e.g. GasNET or Cactus PUGH)	Novice	Intermediate	Expert

Figure 4.6: Frameworks clearly separate roles and responsibilities for a large teams of programmers — enabling computer science experts, numerical algorithms experts, and domain scientists to collaborate together productively.

The framework developer need only provide a standards-based interface that is external to the solver to achieve interoperability. In the commercial market, shallow component interoperability is usually limited within a single vendor's offerings, although some open interchange standards are beginning to emerge based on web services.

- *Deep Component Interoperability:* A few leading HPC laboratories have developed physics component frameworks where the solvers share a common service infrastructure for communications and data management. Physics models can be tightly coupled at this level of interoperability. In this case, the component developer must also heed rules regarding the internal organization of the component in order to achieve interoperability with the framework. This approach hides the complexity of the underlying hardware platform and offers higher-level abstractions for managing parallelism, thereby providing opportunities for improved platform portability and parallel system library optimization by the hardware vendors themselves.

4.8.1 NASTRAN OMD-SA case study

The Open Multi-Discipline Simulation Architecture (OMD-SA) framework [131] is an extensible and flexible Service Oriented Architecture (SOA) for scalable multidisciplinary engineering analysis that has been designed by MSC Software. It supports efficient data transfer for modular multi-physics simulations on HPC systems. Whereas older generation codes used data files to exchange model data between various solvers in a multi-physics application, the OMD-SA architecture enables direct transfers between the components as well as a composition system for combining solver components into a single application. As the simulation data can easily be in the gigabytes or even terabytes, the transfer of data across service layers must be optimized. Specifically, the framework must avoid unneeded copying or transfer of data if it is not absolutely necessary. Services in this framework can be either in a local application space or in a remote application space. Local services should cost

September 14, 2009

Page 31

ECSS Report

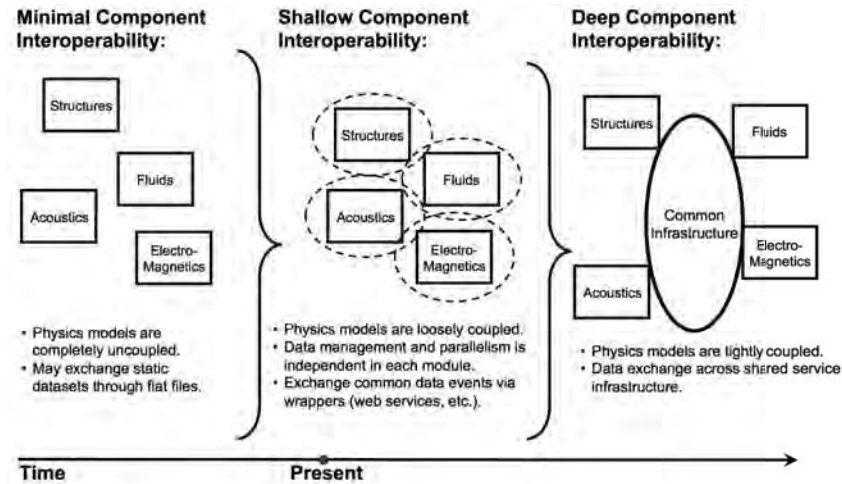


Figure 4.7: Evolution of Software Framework Integration (*From High Performance Computing (HPC) Application Software Consortium (ASC) Summit, March 17, 2008*).

no more than a simple function call so as to ensure that the framework imposes negligible overhead on the overall performance. The framework supports language interoperability so that existing optimized code written in FORTRAN, C, or C++ can be used to implement services within the framework to support this capability. Remote services leverage standard and emerging network protocols to maximize performance. A single service can be used both locally and remotely, and it is up to the framework to determine the usage and the appropriate optimizations relevant to each case. Services must be highly tuned internally for efficient processing, using multi-threading, caching, efficient sharing of memory across services where possible, etc.

There are 4 main elements to the OMD-SA architecture:

- *Component Framework* The component framework is an open SOA model where the services are available on-demand. The component framework is comprised of multiple layers. The services are connected through the Simulation Bus and a common data model that assures scalability and effective application of the services to a simulation application.
- *Simulation Clients* The services are exposed to the various players in the simulation process through different clients, both rich and thin, that address the specific user needs.
- *External Services* External services are available to OMD-SA through standard open plug-in technology. Legacy applications, 3rd party applications, as well as in-house developed applications can be exposed as services to OMD-SA applications.
- *Enterprise Service Bus* The Enterprise Service Bus can be either an existing ESB within an enterprise or a third-party ESB to which OMD-SA will interface. This allows for the use of external enterprise data and processes within a simulation process, *i.e.*, using geometry from PDM within simulation. This allows for the use of simulation services and processes within external enterprise applications.

OMD-SA uses emerging standards for interface definitions and Internet protocols, including OMG-IDL (ISO standard 14750) and WDSL for service description and interface definition, UDDI
September 14, 2009

for service discovery, and SOAP for service invocation/interaction. OMD-SA is an open platform for customers and partners to address extended or proprietary applications through the customizable service APIs, the SOA, and the programmable user interfaces.

OMD-SA is an example of a shallow component interoperability framework in terms of the interfaces it presents to developers. Integrating a component into this framework does not require any changes to the internal data model employed by the solver. Each component is able to share data with other physics solvers using the standards-based APIs to write data to the simulation bus in an operation that looks like writing a file to disk (as would be the case for older multi-physics simulations), but can reside in memory for local data exchanges between simulation clients. All parallelism remains internal to each component, which enables the solvers to be incorporated with little or no changes to their internal data model or data structures. While the shallow interoperability model simplifies coupling of components, the approach does not support any form of abstraction or modularity for the implementation of the parallelism.

4.8.2 Cactus case study

The Cactus Framework [121] is an open source, modular, portable programming environment for collaborative HPC computing. Cactus consists of both a programming model with a set of application-oriented APIs for parallel operations, management of grid variables, parameters etc, as well as a set of modular swappable tools implementing parallel drivers, coordinates, boundary conditions, elliptic solvers, interpolators, reduction operations, and efficient I/O. Although Cactus originated in the numerical relativity community where the largest HPC resources were required to model black holes and neutron stars, Cactus is now a general programming environment with application communities in computational fluid dynamics, coastal modeling, reservoir engineering, quantum gravity and others.

Cactus consists of four main elements:

- The Cactus Flesh, written in ANSI C, acts as the coordinating glue between modules that enables composition of the modules into full applications. Although the architecture is different, the Flesh plays the same role as the Enterprise Service Bus for the OMD-SA framework. The Flesh is independent of all modules, includes a rule based scheduler, parameter file parser, build system, and at run time holds information about the grid variables, parameters, methods in the modules and acts as a service library for modules.
- Cactus modules are termed Thorns and can be written in Fortran 77 or 90, C or C++. Each thorn is a separate library providing a standardized interface to some functionality. The thorns are similar in nature to the Simulation Clients in OMD-SA, but Cactus further externalizes the implementation of parallelism for the thorns, enabling different architecture-specific implementations of parallelism to be plugged in. Thorns providing the same interface are interchangeable and can be directly swapped. Each thorn contains four configuration files that specify the interface between the thorn and the Flesh or other thorns (variables, parameters, methods, scheduling and configuration details). These configuration files have a well-defined language and can thus be used as the basis for interoperability with other component based frameworks.
- Drivers are a specific class of Cactus Thorns that implement the model for parallelism. Each solver thorn is written to an abstract model for parallelism, but the Driver supplies the concrete implementation for the parallelism. For example, the PUGH (Parallel UniGrid Hierarchy) driver implements MPI parallelism, whereas the ShMUGH (Shared Memory UniGrid

Hierarchy) driver provides a shared memory/threaded implementation for the parallelism. The application can use different drivers without requiring any changes to the physics thorns. However, the thorns must be written specifically to the guidelines of the Cactus framework. The modular drivers for implementing parallelism are both the principle advantage of the deeply integrated framework model, but also the most daunting part due to the need to conform to framework coding requirements to take advantage of this capability.

- Cactus modules or thorns are grouped into Toolkits. Cactus is distributed with the Cactus Computational Toolkit that consists of a collection of thorns providing parallel drivers, boundary conditions, scalable I/O etc to support applications using multi-dimensional finite differencing. Community toolkits are provided or are under development by different application areas such as Numerical Relativity and Computational Fluid Dynamics.

The modular design of Cactus with swappable thorns provides several important features:

- Third-party libraries and packages can be used by applications through the abstract Cactus interfaces, decreasing application reliance on any particular package and making it possible to switch to new capabilities as they are available. For example, instead of using the Uni-Grid parallel driver PUGH distributed with Cactus, applications can use a variety of other independent adaptive mesh refinement drivers such as Carpet, PARAMESH, SAMRAI.
- New I/O methods can be added as thorns, and are then available to applications as a parameter file choice.
- Cactus currently supports a variety of output formats including HDF5, NetCDF, ASCII, JPEG, FlexIO, and provides architecture independent checkpoint and recovery along with interfaces for parameter steering and remote visualization.

Cactus has already been shown to scale to large processor numbers (4,000 to 33,000 cores) for different applications, and has active user and developer communities, along with funding from a range of agencies to both improve the infrastructure and build new application areas.

Whereas the shallow component interoperability framework enables modular composition of solver components into a multi-physics application, providing a scalable and modular model for parallelism requires deeper modifications to the code base. Deep component interoperability frameworks such as Cactus and Sierra (discussed in the next section) present an approach where the abstract model for parallel computation is external to each of the components. This requires a larger initial investment in code, but offers additional performance and scalability benefits down the road as systems move towards a massive parallelism on multicore systems.

4.8.3 Sierra case study

Sierra is a software framework [57], which is used for multi-physics computational mechanics simulations primarily targeting finite element and finite volume methods for solid mechanics, heat transfer, fluid dynamics with reacting chemistry, and multi-physics permutations of these mechanics. Sierra is designed around an in-core data model for supporting parallel, adaptive multi-physics on unstructured grids, with an emphasis of simultaneously handling parallelism, dynamic mesh modification, and multiple mesh solutions and transfer operations. Sierra also provides common services and interfaces for linear solver libraries, dynamic load balancing, file input parsing, and mesh file I/O. It was designed to unify and leverage a common base of computer science and data

capabilities across a wide range of applications, and facilitate research, development and deployment of multi-physics capabilities, while managing the complexities of parallel distributed mesh data.

Through its solvers class capability and external interfaces, Sierra provides plug-in capability of a range of solver libraries for different mechanics. Plug-ins play the same role as the thorns in Cactus nomenclature and the Simulation Clients in OMD-SA. At the coupled physics level, Sierra provides a procedural language to support operator splitting methods to couple mechanics, including the ability to iterate to convergence and to sub-cycle physics modules relative to one another. The procedural language, called SolutionControl, allows a user to specify how the coupled mechanics for the various Sierra Regions are executed in sequence, how variables are mapped between the computational domains of each region, and how solution convergence is controlled at the coupling level before moving the simulation forward in time. SolutionControl is the basis for composing solver components into composite multi-physics applications, much as the OMD-SA scripting environment and Cactus Flesh is used to support module composition in those respective frameworks. Sierra also supports limited tighter coupling through forming full Jacobians for multi-physics within a single Sierra Mechanics Region.

Sierra's support for parallelism is pervasive, and is designed to limit the amount of work and complexity associated with parallel data structures for the mechanics developer, so that they can focus on the physics-relevant aspects of their solver module. Like Cactus, the implementation of the parallelism is externalized from each of the solver modules, so that the implementation of the parallelism need not be replicated for each module that comprises the framework. Supporting this capability requires the solvers to adopt some common data structures and conform to framework coding requirements, which is the hallmark of a deeply integrated framework.

4.8.4 Comparison across Frameworks

Examining both shallow and deeply integrated frameworks for modeling and simulation on parallel computing platforms, some common themes have emerged. Physics solvers in these frameworks are implemented as modular software components so they can support flexible reconfiguration for different multi-physics problems. The coupling of physics modules follows loosely coupled at some time step or discrete event as opposed to tight coupling. The framework provides a flexible composition environment that matches the requirements of the application domain. In addition to these common features, deep component interoperability frameworks also partition the implementation of parallelism into separate components, in other words abstracting the implementation of parallelism, to reduce programming errors and support performance optimization and portability across diverse hardware platforms. The key distinction between shallow and deep component interoperability frameworks is that shallow framework components manage their own parallelism and data structures and exchange data using external interfaces, whereas a deep framework components externalize the parallelism and data structures so that they can be optimized and ported independently from the solver component implementations. Our belief is that deep interoperability will increase in importance for extreme scale systems.

Frameworks also provide a base set of services and build tools that simplify the customization of existing software components, and building and integration of new components within the framework. Examples of such services are I/O services, memory management services, error handling services, etc. As existing software modules are to be imported into a framework, their outer layer (a main program calling the subroutines) is peeled off and rewritten as declarations to the framework, which describe the high-level dataflow between the components. The framework manages the coarse-grain dataflow of an application, which is required for efficient parallelization. However,

fine-grain dataflow within subroutines remains under control of the individual components and thus remains highly efficient.

The attraction of shallow integrated frameworks is that they minimize the amount of code rewriting internal to each of the solver components. Each component interacts through a common SOA interface that preserves the opaqueness of the internal architecture of the component. However, such an architecture makes it difficult to impose constraints on the data layouts employed within each module, and therefore can lead to inefficient coupling between components due to the extra layer of data copying that must be employed between components with incompatible data layouts. It also limits the ability of a third party to innovate the implementation of parallelism for the components without getting inside of each module and rewriting the solver implementation. However, the shallow framework component model is well tested in enterprise applications and would require the least amount of effort for ISVs to cooperate. These shallow integration framework architectures consist of a few (tens) of components, each operating on large amounts of data for a significant amount of time. Overheads due to staging, invocation, load distribution etc. are amortized over the run time of the components' activity. One crucial advantage of shallow frameworks is that they arise naturally from preexisting, independent, large software packages as the need for coupling arises. The deeply integrated applications require that solvers agree upon an external data representation for the model data that is exchanged between solvers. This architecture also manages the parallelism external to the solvers. The framework then defines the optimal data layout that is common to all of the components, so as to minimize the amount of data re-copying required to couple components together. In addition, the deeply integrated approach enables the implementation of parallelism to be separated from the solver components, so that innovations in parallelization methods (particularly for multicore processors) can be exploited by the solvers without requiring them to be rewritten. However, the price of such a deep level of integration is that existing solver components must all be rewritten to conform to the frameworks restrictions. This requires a more significant initial investment and a deeper level of cooperation among ISVs, but can lead to a platform that is more scalable to future trends in concurrency.

Deep component interoperability framework architectures consist of many (hundreds) of smaller components, each invoked many times in parallel, operating only on small subsets of the overall data set, supervised by a framework driver layer. Efficiency is guaranteed by the driver layer's control over the data layout, which enables it to orchestrate calculations and relocate data as required. Examples of deeply interoperable framework architectures are Cactus, SIERRA, Chombo, and UPIC. The crucial advantage of deep component interoperability frameworks is the close yet efficient interaction since parallelization is handled by the driver layer, which allows for more accurate multi-physics simulation.

In summary, a tightly integrated framework architecture consists of several key components:

- A backbone which orchestrates the overall simulation, touching only metadata
- A driver layer providing the “heavy lifting”, handling memory management and idiom for parallelism
- Parallel data-layout (domain-decomposition) management components that define or modify the data structures on which the simulation operates (structured, unstructured, AMR)
- Solver components, operating on driver-defined subsets of the data
- Statistics/introspection components, collecting metadata and providing feedback (provenance, performance, progress monitoring)

4.8.5 Domain-Specific Application Frameworks and Libraries

In the following we briefly examine three applications from chemistry that have all departed from norm of generic MPI+OpenMP parallelization and have instead pursued innovative solutions to their computational problems. One enabling characteristic in common is that all three are the result of close and long-term collaborations between application experts and computer scientists.

4.8.5.1 NWChem case study

NWChem was the first quantum chemistry code developed specifically for massively parallel computers and has been funded since circa 1992 by the DOE as part of the Environmental Molecular Sciences Laboratory at Pacific Northwest National Laboratory. Other institutions in Japan, Europe and the USA (*e.g.*, ORNL) contribute to the project and the software is in use at nearly all supercomputer sites worldwide. The genesis of the project was the recognition by Thom Dunning that lack of scalable software precluded chemistry, and in particular environmental molecular science, from exploiting the massively parallel computers emerging in the early 1990's. From the outset, the project adopted a multidisciplinary approach. Molecular electronic structure uses a wide variety of methods to solve a single chemical problem, and the associated computations have complex data structures that can be thought of as block-sparse multi-dimensional matrices. The nature of the sparsity and computation requires careful load-balancing to achieve scalability and the size of the data structures requires distribution to enable computations larger than those possible on a single computer. These challenges led to the development and adoption of the Global Array (GA) library [13] that provides one-sided access to arbitrary blocks of logically-shared but physically distributed matrices. The library has since been extended to other data structures to facilitate its use in many other disciplines. It remains the only portable distributed-shared memory programming library in use since the 1990's.

The combination of one-sided access and data structures/abstractions chosen for the domain revolutionized the writing of scalable applications in chemistry; all scalable chemistry codes either use GA or a derivative of it. Most of the algorithms implemented with GA are inherently more scalable than their MPI counterparts since the one-sided access eliminates unnecessary synchronization, is often closer than message passing to the actual hardware capabilities, and greatly facilitates full dynamic load balancing since computation and data can be easily relocated. GA makes aggressive use of overlapped, asynchronous communication, and also handles the complex translation of addresses from the application (multi-dimension matrix patches) to the hardware (non-contiguous blocks in partitioned linear address spaces). The only components of NWChem still written in message passing are those that for reasons of performance or correctness benefit from the (weak) synchronizations implied by exchange of messages, such as tight coordination of data motion in a parallel FFT or the dependent graph of tasks in a classical matrix factorization.

Going forward, the three main downsides of GA are lack of language support, the fact that its memory model is tied to a specific machine model, and the fact that it only emphasizes two levels of the memory hierarchy (three if disk arrays are considered). The lack of language support is a large source of errors and increased complexity since the user is responsible for tiling accesses to global data structures, managing local arrays, and correctly invoking GA interfaces. GA was consciously designed as a memory model so as to keep its implementation efficient on available computers, but this has now become a major limitation. Within GA, all one can do is move data to/from the computation and this turns out to be insufficient for efficient management/use of more complex data structures. For instance, a distributed sparse tree or hash table would be very inefficient and complex to manage with GA. What is needed is the ability to move computation to data. Finally,

while GA does a good job at managing the coarse grain data motion and parallelism it does little to help with increasing concurrency within SMP nodes. In particular, by forcing the programmer to chose which data to distribute and which to replicate what was once a strength of GA has now locked its existing applications into a specific choice of granularity for their parallelism.

4.8.5.2 NAMD case study

NAMD is a parallel molecular dynamics code designed for high-performance simulation of large bio-molecular systems and was recipient of a 2002 Gordon Bell Award. It is based upon the **Charm++** parallel programming environment and represents a long-standing collaboration between distinguished UIUC researchers Sanjay Kale (Computer Science) and Klaus Schulten (Molecular Biochemistry). **Charm++** is now the only widely used parallel programming model that emphasizes virtualizing all aspects of the computation, with an intelligent runtime taking full responsibility for scheduling and data management. Parallel programs are composed in terms of messaging between objects (chares) addressed in name spaces (chare arrays) without reference to the underlying hardware. This virtualization and separation of responsibilities encourages the expression of the intrinsic parallelism in the application as advocated in Chapter 5. This is exemplified by NAMD being the first chemistry application to execute efficiently on tens of thousand of processors on the IBM BG/L primarily because it was the only application that had expressed sufficient parallelism. Due to its scalability, and the support of Kale's and Schulten's research groups, NAMD is widely used with an increasing amount of science functionality accruing around it.

The main limitation to date of the **Charm++** environment might also be interpreted as a defect of the rest of the world. **Charm++**'s complete virtualization means that its applications are largely incompatible with existing MPI applications unless those applications are also imported into **Charm++** using the AMPI library. There are also performance issues associated with fine grain virtualization and object models that can be alleviated with more powerful language, compiler, and runtime technologies.

4.8.5.3 TCE case study

The Tensor Contraction Engine (TCE) arose out of an NSF ITR project and a DOE SciDAC project, and is the application of compiler optimization and source-to-source translation technology to craft a domain specific language for many-body theories in chemistry and physics. The underlying equations of these theories are all expressed as contractions of many-dimensional arrays or tensors. There may be many thousands of such terms in any one problem but their regularity means that they can be translated into efficient massively parallel code that respects the boundedness of each level of the memory hierarchy and minimizes overall runtime with effective trade-off of increased computation for reduced memory consumption. The approach has been overwhelming successful and now NWChem contains about 1M lines of human-generated code and over 2M lines of machine-generated code from TCE. The resulting scientific capabilities would have taken many man-decades of effort; instead, new theories/models can be tested in a day on a full-scale system. In combination with the OCE (operator contraction engine) that turns Feynman-like diagrams into tensor expressions, the TCE represents perhaps the first end-to-end production quality example of a solution to the semantic gap between applications and hardware.

Clearly, domain specific languages will be an integral part of future computational science and we note that several of the HPCS languages had at their core the idea of being extensible and readily specialized to new fields. However, translating the narrow success of the TCE into broad relevance remains a challenge. For instance, how can application scientists make effective use of

the optimization and compilation tools of computer science without having a computer scientist at their side? What elements are in common between languages tailored to chemistry or material science or linguistics or forestry, and how do we ensure that such programs can inter-operate when composing multi-physics applications?

4.9 Footprints

A “footprint” is the trace of resource usage left by an application as it executes on a system. Understanding such footprints is thus important to understanding whether or when new Extreme Scale systems are in fact capable of supporting the kinds of Extreme Scale systems discussed elsewhere in this chapter. In the remainder of this section, we will examine several different types of footprints.

4.9.1 Application Footprints — System Memory

The quantities of required system memory are highly dependent on specific applications and the associated problem size, architecture balance, research in progress, and anticipated future research. Research includes new-era weak scaling concepts such as multi-scale, multi-physics, new models, interactions, mitigation analysis, data mining, data-derived models, as well as new mathematics and algorithms. Our recommendations are as follows:

- For petascale systems, we recommend $O(100\text{TB})$ to $O(1\text{PB})$ of system memory. These numbers are derived from the following:
 - For all applications, the quantity of system memory should not be less than the bytes/flop/s ratio of the current IBM BlueGene/L (0.083) or BlueGene/P (0.144) systems.
 - Many traditional applications may require a bytes/flop/s ratio similar to Red Storm, Jaguar, and ASCI Purple (0.3-0.5).
 - Applications or data sets may exist that require $O(1\text{PB})$. This is consistent with the bytes/flop/s ratio of 1.0 from Amdahl’s Memory Law (1.0).
- For exascale systems, we recommend $O(10\text{PB})$ to $O(1\text{EB})$ of system memory. These numbers are derived from the following:
 - It may not be possible to perform an exascale application “existence proof” with just $O(1\text{PB})$ of system memory.
 - For many exascale “hero” applications, $O(10\text{PB})$ to $O(100\text{PB})$ system memory may be required.
 - For special applications with massive in-core databases, $O(1\text{EB})$ system memory may be required.

4.9.2 Application Footprints — Storage Capacity

The quantities of required scratch storage are highly dependent on the application and problem size. The amount of scratch storage has traditionally been driven by the need for checkpoint/restart to provide application resiliency — thus it is correlated with the system memory footprint. The need to access additional data in the future may substantially increase the scratch storage requirements due to new-era weak scaling applications. New-era weak scaling may also require that additional data be stored for post-processing data mining. We recommend that scratch storage capacity grow

at a rate faster than system memory. For many applications, scratch storage capacity greater than 10-100x system memory may be required.

- For petascale systems, we recommend O(1PB) to O(100PB) of scratch storage capacity.
- For exascale systems, we recommend O(100PB) to O(100EB) of scratch storage capacity.

The quantities of required archival storage are highly dependent on the application and problem size. The same application requirements that drive scratch storage will drive the need to access additional data from archival storage in the future. We recommend that archival storage capacity grow at a rate faster than system memory or scratch storage. For many applications, massive archival storage capacity greater than 100x system memory may be required.

- For petascale systems, we recommend greater than O(100PB) of archival storage capacity.
- For exascale systems, we recommend greater than O(100EB) of archival storage capacity.

4.9.3 Application Footprints — Node/System Memory Bandwidth and Latency

”Local” memory bandwidth and latency requirements will be driven by the fact that new applications will have reduced vertical locality on a node because the trend in new applications are to employ:

- New mathematics and algorithms that trade regular data access for improved convergence.
- Model-directed adaptive mesh refinement (AMR) for improved computational accuracy where required.
- Data-derived models, data-mining, and interaction studies in multi-physics models.

It will be a challenge for hardware designers to provide adequate bandwidth and sufficiently low memory latency to keep processors “busy”. There must be adequate memory bandwidth to feed instructions to the processors/cores on a node (given the high latencies). Key techniques include attacking the traditional “Memory Wall” and employing latency tolerance techniques with massive multi-threading.

”Global” memory bisection bandwidth and latency requirements will be driven by the fact that new applications will have reduced “horizontal” locality for many of the same reasons that there will be reduced vertical locality in new applications. Bisection bandwidth requirements will be highly dependent on application characteristics such as:

- Scientific and Engineering Codes including solvers for Partial Differential Equations (PDEs) and 3-D meshes
 - Structured Grids — nearest neighbor communications
 - Unstructured Grids — indirect addressing and random communications
 - Adaptive Mesh Refinement — move extensive amounts of data around machine
- Multi-scale, multi-model, etc. — less likely to be able to map data/processes to (nearest-neighbor) locations to minimize communications
- Models may trade global data access for better convergence in new mathematics and algorithms

- Application reliability could have a significant impact on bisection bandwidth in the future

Global memory latency will be driven by these same application characteristics. Given the physical size of exascale computers, system diameter and “speed-of-light” issues will set the lower-bound on latency. Hardware designers must provide sufficiently low latency or adequate bandwidth and latency tolerance in keeping with Little’s Law [14, 15].

After analyzing these application characteristics we recommend that latency be as low as possible and the bandwidth be as follows:

- For petascale systems, we recommend bisection bandwidths — $O(50\text{TB/s})$ to $O(1\text{PB/s})$. These numbers are derived from the following:
 - For all applications, bisection bandwidth should be no less than current 3-D topologies scaled to a petaflop/s performance rates.
 - * Sample XT4 configuration (40 x 32 x 24) @ 318Tflop/s \leftarrow 19.4TB/s
 - * Scaled to (80 x 64 x 48) @ 2.4Pflop/s \leftarrow 80.0TB/s
- For many applications: bisection bandwidth may need to approach the bandwidth specified in the DARPA HPCS program (500.0TB/s-3.2 PB/s)
 - For exascale systems, we recommend bisection bandwidths — $O(10\text{PB/s})$ to $O(1\text{EB/s})$. These numbers are derived from the following:
 - While it may be possible to perform an exascale “existence proof” at $O(1\text{PB/s})$, real applications will require substantially greater interprocessor communications capability.
 - For nearly all exascale applications, bisection bandwidth should scale with the quantity of memory — $O(10\text{PB/s})$ to $O(1\text{EB/s})$.

4.9.4 Summary — Design Sweet Spots

In summary, we do not expect applications to strongly scale three or more orders of magnitude efficiently to provide faster application run times with similar quantities of system memory. Some applications will be able to scale to petascale and on to exascale using traditional weak scaling. We anticipate that many applications to be run at exascale will employ new-era weak scaling and employ one or more of the following: Multi-scale, Multi-physics (multi-models), New models, Interactions, Mitigation analysis, Data mining, and Data-derived models.

In figure 4.8, we present a summary chart of the petascale and exascale application derived “footprints”. On this chart, we have proposed design “sweet spots” that will address the capabilities of a majority of new exascale applications. These footprints and “sweet-spots” are in agreement with the 2007 Exascale Hardware study [62].

4.10 Two Illustrative Graph Scenarios

An examination of two graph algorithms illustrates some of the issues with exascale software. An N-body code (*e.g.*, GROMACS) in which neighbor lists of vertices are traversed to compute interactions represents one data point where the locality is easily managed. A shortest-path algorithm, on the other hand is more challenging because of fine-grain mutability and lower arithmetic intensity.

Exascale Software Study

	Petascale		Exascale	
	Range	"Sweet Spot"	Range	"Sweet Spot"
Memory Footprint				
System Memory	O(100TB) to O(1PB)	500 TB	O(10PB) to O(1EB)	100 PB
Scratch Storage	O(1PB) to O(100PB)	10 PB	O(100PB) to O(100EB)	2 EB
Archival Storage	Greater than O(100PB)	100 PB	Greater than O(100EB)	100 EB
Communications Footprint				
Local Memory Bandwidth and Latency	Expect low spatial locality			
Global Memory "Bisection" Bandwidth	O(50TB/s) to O(1PB/s)	1 PB/s	O(10PB/s) to O(1EB/s)	200 PB/s
Global Memory Latency	Expect limited locality			
Storage Bandwidth	Storage bandwidth will need to grow at a faster rate than system peak performance or system memory growth			

Figure 4.8: Petascale and Exascale Application Design "Sweet Spots"

4.10.1 N-Body

N-body codes are used to study the interactions of systems ranging from proteins to galaxies. N-body codes are often structured as graph algorithms where each particle maintains a neighbor list of other particles within an interaction radius. Each time step, every particle computes an interaction with every neighbor, and this interaction is used to update the state of the particle at the end of the time step. Every few timesteps (typically 10) the neighbor lists are recomputed often using a cell structure that reflects the three-dimensional nature of the problem.

In very rough terms the code is:

```

for each timestep {
    if(neighborListsStale()) {
        forall particle in particles {
            particle.neighbors = computeNeighbors(particle) ;
        }
    }
    forall particle in particles {
        forall neighbor in particle.neighbors {
            computeInteraction(particle, neighbor) ;
        }
    }
    forall particle in particles {
        updateState(particle) ;
    }
}

```

This is a gross oversimplification, but captures many important issues outlined below. In particular, the analysis assumes that the particles in a cell interact only with a thin boundary layer of particles

in adjacent cells and that the partition of force computations follows the partition of the particles.

Parallelization: The bulk of the computation has parallelism of PN where P is the number of particles and N is the number of neighbors per particle. The update step has parallelism of P but accounts for a much smaller fraction of the total computation. For a large problem, there can be 10^8 particles each with 10^3 neighbors, so the amount of parallelism may be 10^{11} .

Note that this parallelism is needed both to take advantage of multiple cores and to hide latency to higher levels of the storage hierarchy.

Synchronization: The required synchronization is implied by the program data flow. Each time step must use the updated state from the last time step. There is no need for two barriers per time step although many implementations would over-synchronize the application in this manner. A looser synchronization would give improved performance.

Locality: The amount of locality depends on the underlying graph. If the graph is generated by connecting all particles within an interaction radius in a 3D space, it will have $X^{(2/3)}$ connections out of a partition of X particles. A partition of X particles can be gathered into a local memory along with a halo of neighbor particles and operated on entirely locally. XN operations are performed for each $X + \text{halo}(X)$ particles fetched. The size of a partition X is driven by the amount of storage available at a level of the storage hierarchy. This partitioning is done recursively down the hierarchy. Achieving this locality depends on generating a good (min-cut) partition of the set of particles P . Because the graph changes over time, this partition also changes over time.

A related issue is the computational intensity of the program — how much work is done in the routine “computeInteraction”? For molecular dynamics codes, the interaction is complex — 100s of operations — giving a high arithmetic to bandwidth ratio.

Load Balance: Load balancing can be accomplished by distributing partitions to levels of the hierarchy — nodes, multi-core chips, regions on these chips, and cores. This distribution can be dynamic — and needs to be at least partly dynamic as the partitions will change as the graph is modified. As opposed to fine-grain work stealing, load balancing needs to be done at a coarser grain — that of partitions — to avoid destroying locality.

In summary, the N-body problem has lots of parallelism and easily exposed locality. It is straightforward to load-balance and synchronize the application provided the parallelism and locality can be easily expressed.

4.10.2 Shortest Path

Consider a single-point shortest path problem in a graph. Starting at a single point, for each vertex in an active set, we visit all neighbors, possibly updating their distance, and if updated we add the neighbor to the active set. In rough form the code looks like:

```
activeSet = Singleton(sourceNode) ;
while(notEmpty(activeSet) {
    for vertex in activeSet {
        for neighbor in vertex.neighbors {
            test = Update(vertex, neighbor) ;
            if(test) Insert(neighbor, activeSet) ;
        }
    }
}
```

Again, this is an oversimplification. For example, the active set should be managed as a priority queue, but it captures some interesting behavior.

September 14, 2009

Page 43

ECSS Report

Parallelism: The amount of parallelism depends on the shape of the graph. It starts with a single thread at the source node and increases as the active set increases. Depending on how many times a node is revisited, the average parallelism is roughly V/D where V is the number of vertices and D is the diameter of the graph. For a graph of 10^9 nodes with diameter 100, the average parallelism would be about 10^7 .

Synchronization: The calls to the routine “Update” need to be atomic. Multiple vertices may have the same neighbor, and one update of that neighbor must be completed before the next is started. The difficulty of this atomic action depends greatly on implementation — specifically where the update is performed. If the update is done on the core that is co-located with neighbor, then this can be done relatively inexpensively. The atomic section of code can be run out of local memory with no long latency accesses or message round trips.

On the other hand, if one attempts to run Update on an arbitrary node, the synchronization can become prohibitively expensive. This requires acquiring a lock (or the equivalent), fetching the current value of neighbor, writing back the updated value, and then releasing the lock. This requires at least six messages (three round trips), and the number can easily be several times this number if a cache coherence protocol is involved. The amount of time a neighbor remains “locked” is a critical parameter here as it affects the amount of usable parallelism.

Locality: Locality depends on the nature of the graph and the contents of the active set at a given point in time. However, it is likely to be low. The active set represents a slice of the graph at a given distance from the source and hence is not likely to be highly interconnected. The re-use is most likely about the average degree of a vertex — since each vertex gets updated by each of its neighbors. To get this amount of locality, one needs to partition the active set so that vertices that share neighbors are in the same partition (which may not be easy).

The locality problem is made worse by the fact that the update function is likely very simple — a few arithmetic operations — making the arithmetic to bandwidth ratio low. Despite the low locality, we can make data transfers efficient by doing block transfers (gathers) of partitions of the active set and the neighbors of that partition. Neighbors that are shared by partitions of the active set are placed in only one partition.

Load Balance: As above, this can be load balanced by distributing partitions of the active set to levels of the storage hierarchy. The load balancing needs to be done at the level of partitions to avoid destroying what little locality there is.

Chapter 5

Challenges in Expressing Parallelism and Locality in Extreme Scale Software

The focus of this chapter is on the challenges in expressing parallelism and locality that are encountered by application-level programmers across the three classes of Extreme Scale systems. The task of managing the parallelism and locality is relegated to the system software discussed in Chapter 6. It is likely that some heroic programmers, particularly for the data-center and embedded configurations, will wish to program directly at the system level using the interfaces in Chapter 6. However, for the remainder, it will be critical to address the challenges outlined in this section to enable them to use the capabilities of Extreme Scale systems. The capabilities described in this chapter are intended to address the needs of Applications (Chapter 4), serve as a portable interface to Runtime Management of Locality and Parallelism (Chapter 6), and provide information to Extreme Scale Tools (Chapter 7).

5.1 Application Programming for Extreme Scale Require Fundamental Breakthroughs

As outlined in Chapter 4, if applications are to be able to tap the power of future extreme-scale systems, they must exploit parallelism at multiple scales, at fine granularity, and across a wide variety of irregular program structures, data structures, program inputs, and in widely varying dynamic resource environments. In short, there are fundamental challenges which amount to a *major crisis* in the underpinnings of software development for all high performance computing systems. Simply put, the existing programming approaches we have relied on to get to 100,000 fold parallelism in nascent petaflop computing systems of today will not take us to extreme scale. They are too rigid and labor-intensive, while also failing to expose sufficient parallelism and locality to enable scalability and portability to future extreme scale computing systems.

Simultaneously expressing all available application parallelism and locality is a significantly different task from writing traditional sequential programs. Sequential programs can be thought of as a single expression of operation order and implied data movement (and thereby realized locality), and traditional program optimizers and parallelizers required “proof” of effect equivalence to make limited changes around that sequential order for performance [37], parallelism [85,90,118,145] or locality [85,119,144]. With expression of only a single path, the limits of analysis meant

that the computation expressions were over-constrained. While those programming systems could increase parallelism and performance moderately, they are incapable of taking sequential application programs and automatically creating the needed parallelism and locality for efficient extreme scale parallel execution. Expressing all available application parallelism and locality requires a much richer expression — of the lesser constraints required to realize the computation, and the rich structure of locality over which the underlying program implementation system can play to achieve low power and high parallelism — both being synonymous with performance in extreme scale computing.

We identify several critical challenges to enable applications to exploit parallelism at multiple scales, at fine granularity, across a wide variety of irregular program structures, data structures, program inputs, and in widely varying dynamic resource environments:

- Billion-fold *parallelism* is required to tap the performance of extreme scale machines; achieving this requires flexible exploitation of regular and irregular parallelism across a range of scales from coarse to fine-grained.
- *Locality* is a critical requirement both to reduce energy per unit computation, and reduce latency (which in turn reduces the energy and complexity costs of managing many concurrent outstanding memory operations). This is a critical requirement (not an optional one) because the power requirements of extreme scale systems are directly tied to achievable performance [62].
- A third critical requirement is for the programming system to provide a simple *execution model* for the programmer to think about. As the scale and complexity of software to meet a variety of mission and commercial needs continues to expand in complexity, a simple execution model will reduce the application programming complexity required to achieve the goals of exposing all parallelism and locality. While the execution model may be defined at a high level of abstraction, the underlying programming system should enable programmers to provide non-binding guidance (hints or “default” control) when needed, thereby providing a level of *performance transparency*.

While we do not expect any one means of balancing these factors to be appropriate for all extreme scale applications, we believe that significant progress is required (and possible) in all three. Attacking all three simultaneously is the grand challenge of extreme scale programming. These challenges are demanding, but they must be addressed if extreme scale systems are to achieve their performance potential.

5.2 Portable Expression of Massive Parallelism

The requirement of pervasive extreme-scale parallelism outlined earlier demands that all of the intrinsic parallelism be exposed at all levels in the application. This is a marked contrast to current practice where programmers repeatedly rewrite applications to expose incrementally more parallelism for the next generation of hardware. Instead, our goal should be to express all opportunities for parallelism, leaving the choice of what to exploit to the layers of the software stack responsible for managing parallelism and locality (Chapter 6). While this is likely to be a more demanding task for current programmers who have been trained in sequential programming, it is expected that expression of parallelism and locality will be simplified in future programming models that *break sequential habits of thought* [132]. In this section, we briefly summarize some of the key points made in [132] and related work.

September 14, 2009

Page 46

ECSS Report

First, a major focus in writing efficient sequential code is to *minimize the total number of operations*. In contrast, efficient parallel code needs to focus on maximizing parallelism *i.e.*, *minimizing the number of operations on the critical path*. With modern memory hierarchies, both sequential and parallel code must also focus on *improved locality*, but parallel code offers more opportunities than sequential code to (say) perform redundant operations to reduce communication. As mentioned earlier, locality optimization will have a first-order impact on energy reduction for future Extreme Scale systems. Second, good sequential algorithms attempt to minimize space usage and often include clever tricks to reuse storage; however, parallel algorithms need to use extra space to permit temporal decoupling and to achieve larger scales of parallelism. Finally, sequential idioms often stress linear problem decomposition through sequential iteration and linear induction. On the other hand, good parallel code usually requires multi-way problem decomposition and multi-way aggregation of results. A simple example is the difference between specifying a summation as a sequential iteration vs. a Fortran 90 SUM intrinsic for arrays.

A fundamental issue in the portable expression of parallelism is the need for data structures that lend themselves naturally to *data-parallel* operations. Fortunately, the *array* or *vector* data structure, which is a cornerstone of traditional HPC applications, is very well suited to data parallelism as evidenced by programming languages such as APL [82], Fortran 90 [102], NESL [42] and Ct [67]. These languages are able to express both flat data parallelism on vectors (element-wise operations, reductions, constrained permutations) and nested data parallelism on sparse or indexed vectors. *Streams* represent another data structure that is well suited for parallelism, as exemplified in data flow languages and programming models such as Sisal [100], Synchronous Data Flow [93], Brook [45] and StreamIt [69]. However, graph and other pointer-based data structures necessary for new Extreme Scale applications pose additional challenges for expression of parallelism and locality. The notion of *abstract collections* in modern object-oriented languages can help bring some of the benefits of data parallelism from arrays and streams to pointer-based data structures. In addition, *asynchronous dynamic parallelism*, as embodied in languages such as Cilk [43], Chapel [53], Fortress [38], and X10 [50], is necessary for operating on irregular data structures. Compared to current approaches, a key challenge for Extreme Scale is the ability to express this parallelism at the finest granularity possible, while delegating to the implementation the choice of what parallelism to exploit in a locality-sensitive manner.

In summary, a program that is organized according to sequential thinking and linear problem decomposition principles will be very hard to parallelize, whether by manual or automatic means. On the other hand, a program organized according to parallel problem decomposition principles should be easily run either in parallel or sequentially, according to available resources. At first, the costs and overheads for the “intrinsically parallel” approach may be daunting, but we have no choice than to overcome these challenges so as to enable software to use future Extreme Scale systems. Along with advancing foundational technologies for intrinsic parallelism and locality, we will need to also advance the pedagogy and curricula for software development. We will need to teach new strategies for problem decomposition ranging from data structure design to algorithmic organization that do not incorporate any inherent sequentiality. Approaches to multi-way problem decomposition may make the process of combining general sub-solutions harder than the sequential case, but this is our only hope for program portability in the future.

5.3 Portable Expression of Locality

The term “locality” refers to the logical or physical proximity of data to the units that perform computation on them. As mentioned earlier, locality optimization will have a first-order impact on

energy reduction for future Extreme Scale systems. Also, good sequential algorithms attempt to minimize space usage and often include clever tricks to reuse storage; however, parallel algorithms often require extra space to enable the temporal decoupling necessary for larger scales of parallelism. In Chapters 3 and 4, we also discussed the need for extra storage in weakly scaled parallel algorithms.

Programs must capture the opportunities for reuse and locality independent of machine structure or management policy. The compiler and runtime/OS can then decide how to best exploit the locality that has been exposed — fitting it to the structure of a target machine. Locality must include both *horizontal* (between processing elements) and *vertical* (between levels of the hierarchy) types.

5.3.1 A Simple Example

Consider the following simplified pseudocode fragment from a fluid dynamics application:

```
for t in timesteps {
  forall cell in cells {
    forall neighbor in cell.neighbors {
      compute_flux(cell, neighbor) ; // uses old value of pressure
    }
  }
  forall cell in cells {
    compute_pressure(cell) ; // uses both x-flux and y-flux
  }
}
```

Here `cells` is a possibly irregular collection of cells that is large enough so that it fits only in the aggregate main memory of a large machine. The computation iterates over the collection twice. On the first pass, it computes the fluxes through each face of a cell as a function of its pressure and that of its neighbors. On the second pass, it computes the pressure of each cell as a function of these fluxes. Each cell structure holds a collection of pointers to the cell's neighbors, the fluxes through each face of the cell, and the pressure within the cell.

The locality of this program is expressed by the neighbor relationship among cells, which also captures the dependences in the computation. The neighbor graph is data dependent, and may be time varying. Ideally we would like to exploit this locality by capturing both the reuse of shared neighbors and the producer-consumer locality between flux and pressure. Both of these forms of locality can be exploited horizontally and vertically.

While the neighbor relationship captures the locality of this program, it is easier to exploit this locality if it is expressed by partitioning the cell collection into sub-collections of arbitrary size in a manner that minimizes the number of external neighbor links from each sub-collection. Consider the following pseudocode:

```
partition(N, cells, parts) {
  forall part in parts {
    forall cell in part {
      forall neighbor in cell.neighbors {
        compute_flux(cell, neighbor) ; // uses old value of pressure
      }
    }
  }
}
```

September 14, 2009

Page 48

ECSS Report


```
forall part in parts {
  forall cell in part {
    compute_pressure(cell) ; // uses both x-flux and y-flux
  }
}
```

Here we assume an application-specific function decomposes the cell collection into a partition called *parts*. Each *part* can then be mapped to a given *node* to exploit horizontal locality or to a level of the memory hierarchy to exploit vertical locality. Multiple levels of vertical locality can be expressed in this manner by recursively partitioning a collection. In a style motivated by Sequoia [64] and by Hierarchical Place Trees [147], we can write:

```
void compute_cells(cells, level) {
  if(is_leaf(level)) {
    forall cell in cells {
      // do the computation
    }
  } else {
    partition(N[level], cells, parts) ;
    forall part in parts {
      compute_cells(part, level-1) ;
    }
  }
}
```

To capture producer-consumer locality, we need to fuse the two forall nests as shown below:

```
partition(N, cells, parts) {
  forall part in parts {
    forall cell in part {
      forall neighbor in cell.neighbors {
        compute_flux(cell, neighbor) ; // uses old value of pressure
      }
      compute_new_pressure(cell) ; // but don't update old pressure yet
    }
  }
  make_new_current() ; // now make new pressure visible
}
```

The programmer here is expressing the producer/consumer locality between fluxes and pressures by bringing the two computations closer together. Now the fluxes are used (consumed) as soon as they are generated (produced). Hence the flux values can be captured in the smallest level of the storage hierarchy and never have to be written out to main memory. This transformation also improves reuse as it avoids having to read the cell back in to compute its pressure. To avoid creating a read-after-write (RAW) hazard, however, the update of pressure needs to be synchronized so that all flux computations in the current timestep see the old pressure value. While in theory such transformations could be automatically discovered, it is much easier if this locality is explicitly expressed, rather than leaving it to be discovered.

5.3.2 Parameterized Decomposition

As shown in the example above, the key to expressing locality in a portable manner is a parameterized decomposition of a dataset. The `partition` function above expresses what the programming system needs to know about the collection to exploit both horizontal and vertical locality. Specifying the partition function is a key aspect of expressing locality.

The partition function is application specific. For dense matrices, partitioning is just a block decomposition. For graphs, the structure of the problem may be exploited to generate good partitions inexpensively. For example, in a 3D code, the physical partitioning of space into contiguous 3D regions may be used to generate good partitions.

The temporal nature of the partition is also application specific. For some codes the partition is data independent. For others it is data dependent but static *i.e.*, the partition must be computed when the data set is loaded, but then remains constant for the remainder of the program execution. In other cases, the partition is time varying — the partition changes (perhaps incrementally) each timestep.

Expressing locality also requires that the dependence graph of the application be easily determined from the source code. In the example above, the dependence information is easy to determine — cell fluxes depend on cell and neighbor pressures and cell pressure depends on cell fluxes. For applications with significant indirection, such dataflow can be harder to determine making it more challenging for a compiler and run-time system to discover locality. Programs must capture the opportunities for re-use independent of machine structure or management policy, let the runtime/OS exploit this as it can.

5.4 Portable Expression of Synchronization with Dynamic Parallelism

Writing programs using today's state-of-the-art synchronization primitives is akin to using assembly language for programming. All the burden of performance, scalability and correctness falls squarely on the shoulders of the programmer with minimal support from the programming languages, development tools, runtime systems or hardware. In order to make parallel programs robust, portable, and scalable and reduce the burden on the programmers, many innovations are required in synchronization and communication. As an example, the *phasers* construct [124, 125] extends X10's clocks [50] so as to integrate collective and point-to-point synchronization with fine-grained dynamic parallelism, while providing a *next* statement that guarantees that all synchronizations will be performed in a deadlock-free manner. Each fine-grained task has the option of registering with a phaser in *signal-only/wait-only* mode for producer/consumer synchronization or *signal-wait* mode for barrier synchronization. Support for dynamic parallelism dictates that it should be possible for new tasks to be dynamically added and dropped from phaser registrations, which creates a potential challenge to avoid race conditions between synchronization operations and registration add/drop requests. The fine-grain synchronization that accompanies fine-grain parallelism also presents the challenge of *phaser contraction* [126] to reduce the synchronization overhead when reducing the actual parallelism that is exploited on a given system.

One of the biggest challenges with synchronization for a programmer is the difficulty in avoiding deadlock and data races, both of which can appear non-deterministically in current programming models. Of the two, data race avoidance is more challenging than deadlock avoidance, since deadlock freedom can be enforced by well-defined programming practices and the use of deadlock-free programming constructs such as transactions and phasers. Removing non-determinism from

the programming model (as in declarative and functional programming approaches) can greatly simplify the testing and debugging of parallel programs, but the key challenge there is to ensure that the resulting model is sufficiently expressive for Extreme Scale software while still being efficient enough for execution on Extreme Scale hardware.

5.5 Support for Composable and Scalable Parallel Programs with Algorithmic Choice

Sequential idioms often stress linear problem decomposition through sequential iteration and linear induction. On the other hand, good parallel code usually requires multi-way problem decomposition and multi-way aggregation of results. A simple example is the difference between specifying a summation as a sequential iteration vs. a Fortran 90 SUM intrinsic for arrays. Extreme scale software will tax our ability to build programs — due to the complexity of application logic, sophisticated irregular algorithms, and program structure required. To manage these complexities, it must be possible to tap the full collection of higher level programming tools available to modern software engineering. In particular, in extreme scale programming systems, the expression of parallelism, concurrency and synchronization, and locality must interact gracefully with modern software engineering tools. The importance of composition was discussed earlier in Section 4.8. Ideally, concurrency, synchronization, and locality would be expressed and managed at the level of programmer meaningful abstract entities — objects or data abstractions — not individual bytes or words or values. (Of course, the system level interfaces outlined in Chapter 6 may well operate on lower-level machine-specific primitives and datatypes.)

To reduce the development cost of extreme scale software, it is imperative to have programs that can seamlessly scale across embedded, departmental and data center-sized extreme scale systems. However, it is often impossible to obtain a one-size-fits-all solution for high performance algorithms that will work effectively in both ends of the scale. Often the best algorithm for an architecture is tightly coupled to differences in parallelism, communication, and available system resources. Different algorithms are effective at different sets of architectural parameters. Current compiler and programming languages are unable to handle algorithmic choice. Thus, it is important to have programming languages and compilers that also support algorithmic choice. These systems should let the programmers express different algorithms to solve a problem and have the compiler and runtime system automatically identify the best algorithm for the given deployment.

5.6 Managing Heterogeneity in a Portable Manner

With the advent and increasing popularity of hybrid architectures, programming systems face the challenge of how to efficiently exploit multiple levels of parallelism, often coupled with different memory systems, instruction sets, or even numerics. In current-day systems, such as the LANL Roadrunner system [86], there may be as many as three distinct types of processors with distinct memory, messaging, and performance characteristics. These elements of heterogeneity are managed explicitly by the application programmers — through the use of coroutine-style models (one for each type of heterogeneity), explicit message passing for data movement, and distinct address spaces. Other examples of heterogeneous systems might include instruction set and performance heterogeneity or simply differences in memory structure such as cache coherent shared-memory, partitioned global address space, or shared-nothing. Unfortunately, if such characteristics of hardware heterogeneity are explicitly addressed by the programmer, not only is the programming effort

increased, it is likely that the software will not be functionally portable, much less performance portable to other systems.

Extreme Scale systems of the future may have both *designed heterogeneity* (in dimensions such as architecture, organization, instruction set that are exhibited today in hybrid systems), as well as *intrinsic heterogeneity* that arises from manufacturing variability, configuration, or aging differences. It is critical the software built for such large-scale parallel systems address the heterogeneity of the system in a fashion that supports portability of the applications. That is, it should be possible to move applications from one machine to another — with different heterogeneous characteristics — without significant change at the application source code level. This imposes major challenges in expression of parallelism, locality, and computation so as to both enable the compiler and runtime to deliver performance on one Extreme Scale system, but also in a form portable and flexible enough that it can enable the compiler and runtime to deliver performance on other heterogeneous Extreme Scale systems. This is a daunting challenge, but is in our view a fundamental requirement for a technology landscape that supports Extreme Scale computing.

Chapter 6

Challenges in Managing Parallelism and Locality in Extreme Scale Software

Earlier in this report, we summarized the hardware characteristics of future Extreme Scale systems (Chapter 2) as well as the challenges involved in developing applications (Chapter 4) and expressing parallelism and locality (Chapter 5) for such systems. In this chapter, we focus on the challenges and implications in *software management of parallelism and locality* for Extreme Scale *i.e.*, in bridging between high-level application frameworks and programming models and the realities of Extreme Scale hardware. Current software for high-end data-center, departmental and embedded systems build on a *classical software stack* which primarily consists of operating systems, parallel runtimes, static compilers, and libraries. Different embodiments of the classical software stack have been used in the past for data-center-class capability systems, departmental systems, and embedded systems with thinner and more restricted stacks used at the two extreme ends, and a richer software stack used for departmental systems in the middle. However, as described in the following sections, the general structure of the classical software stack has remained largely unchanged for decades, and will be highly mismatched to the requirements of all three classes of future Extreme Scale systems.

6.1 Operating System Challenges

6.1.1 Introduction

Extreme Scale processors containing hundreds or even thousands of cores will challenge current operating system (OS) practices. Many of the fundamental assumptions that underlie current OS technology are based on design assumptions that are no longer valid for a Extreme Scale processor containing thousands of cores. In the context of Exascale system requirements, as machines grow in scale and complexity, techniques to make the most effective use of network, memory, processor, and energy resources are becoming increasingly important. In its role as gate-keeper to all these resources, the OS becomes a major obstacle in allowing the application to view the hardware in accordance with the Extreme Scale Execution Model outlined in Section 3.1. A baseline challenge for the exascale software stack is: how to reduce OS overheads without compromising the need to protect hardware state from errant or malicious software.

Execution models that support more asynchrony will be necessary to hide *latency*. Such execution models will also require more carefully coordinated scheduling to balance resource utilization

and minimize work *starvation* or resource *contention*. These execution models will also require extraordinarily *low-overhead*, fine-grained messaging. However, the attributes required by the execution model are nearly impossible to achieve when the OS intervenes for every operation that touches its privileged domain *e.g.*, for exclusive and privileged control of scheduling policy, for exclusive ownership of resource management policies, and for inter-processor communication operations.

6.1.2 What is wrong with current Operating Systems?

Over time, operating systems have evolved into multifaceted and hugely complex software implementations that have accreted a broad range of capabilities. We refer to the challenge of breaking the OS apart based on separation of concerns as “deconstructing the OS”. Below are a subset of leading issues that motivate the need to reexamine the underlying assumptions that are encoded in current OS implementations. These issues are discussed from the viewpoint of all three classes of Extreme Class systems, not just the data center class for which specialized solutions have been developed in past work on developing lightweight kernels for supercomputers.

6.1.2.1 Time Sharing

Time-sharing OS’s are built around the assumption that the CPU is a precious resource that must be shared. This is no longer true for a CMP (Chip level Multi-Processor) containing thousands of cores and constrained memory bandwidth.

- Old Conventional Wisdom (CW): When CPUs are considered the most precious resource, time-multiplexing is performed to share access. However, when hundreds of CPUs are available, it no longer makes sense to suffer the overheads incurred by context switching. Indeed, the cost of a context switch is not merely the time spent preserving registers because the associated cache pollution (and other shared resources) can substantially reduce CPU throughput. Context switching makes inefficient use of the new precious resources, which are energy, on-chip memory and off-chip bandwidth.
- New CW: If cores are cheap, then allocation of cores should be spatially partitioned for function rather than offering time slices of a single resource. This spatial partitioning is analogous to Logical Partitioning (LPAR) from 1970’s era mainframe terminology.
- Research examples: MITOSYS (MIT/Berkeley), K42.

6.1.2.2 Inter-Processor Communication

All device interfaces are at the OS’s privilege level in a typical system of today. Therefore, any access to virtualized device interfaces is mediated by the OS in order to protect the hardware interface. However, the overhead of the privilege change and additional data buffer copies required for OS mediation has given way to a wide variety of “OS bypass” and “user space messaging” implementations such as VIA, PORTALS, and DRI. However, these approaches expose the hardware to irrecoverable state corruption by errant software. Whereas current OS works in-band to protect the hardware state, it would be more efficient to employ extra cores or some other supervisory hardware to monitor application-to-device transactions out-of-band to check for state corruption without adding additional overhead to the communication stream.

- Old CW: invoke OS for *any* interprocessor communication or scheduling to protect hardware state from corruption.

- New CW: direct HW access allowed by application, but hypervisor monitors for state corruption out-of-band from the transactions (perhaps using a dedicated subset of cores as observers).
- Research examples: Singularity, MVIA/MVAPICH. that is isolated to OS bypass for MPI)

6.1.2.3 Interrupts and Asynchronous I/O

Background task handling for asynchronous operations are currently handled by threads and signals in modern OS's and application designs. In particular, devices must interrupt the CPU in order to invoke the device driver software to service their requests.

- Old CW: Interrupts and threads (a side-effect of time-multiplexing access to a core) are used to implement asynchronous operations or system calls in user codes. Even more fundamentally, device handling is implemented by interrupting the CPU for the time-critical top-half of the device driver and then scheduling the bottom half of the driver to run on the next available time-slice of the CPU. The interrupt subjects processes to non-deterministic delays that can result in load-imbalances for parallel applications.
- New CW: If CPUs are abundant, side-cores can be dedicated to asynchronous I/O and device handling. For that matter, cores can be used to implement programmable DMA and asynchronous I/O instead of using dedicated hardware components. Using a core or background thread to implement DMA eliminates the need to write to the device control interface for a dedicated DMA engine, which require costly `msync()` operations by the requesting CPU.

In this way, finer-grained spatial deconstruction can be achieved with legacy device drivers wrapped and isolated on separate cores, interrupts delivered automatically to free cores, and traditional facilities (*e.g.*, file systems) handled as servers on separate cores. In addition, resource allocation and Quality of Service (QoS) guarantees for network and memory bandwidth and fair access to I/O modules can be obtained by using hardware mechanisms for QoS as well as software-based policy implementations. The complexity added, however, is in duplicating management information of virtual memory and other resources.

6.1.2.4 Device Drivers and Virtualization

OS's play an important role in virtualizing finite hardware device interfaces — making it appear to each application that it is the exclusive owner of a replicated copy of the device interface. In the parallel context, current OS's assume a workload of uncoordinated processes that stochastically vie for control of finite/virtualized devices.

- Old CW: OS's currently implement a greedy allocation policy where the first process acquires a lock to gain exclusive access to a device (or OS/driver interface) for each I/O transaction. This approach is sensible for stochastic access to the virtualized resource, but very bad for highly-synchronous parallel algorithms. Resource and lock contention hurts performance by flooding the inter-processor communication network with redundant/spinning lock acquisition requests. Locks also serialize otherwise parallel processes when they attempt to access the same resource (such as the network interface), and subjects them to nondeterministic delays.
- New CW: A new OS will need to use a QoS management for symmetric device access by a large number of entities, or hand over coordinated scheduling of device access to the application. The development of novel mechanisms for coordinating parallel access to finite number of virtualized device interfaces is essential.

- Examples of Research: NOOKs, K42.

6.1.2.5 Fault Isolation

Another important role of operating systems on more robust systems is fault isolation. This is particularly difficult for large SMP systems as errors can propagate rapidly through the system and are extremely difficult to track down. With growing node concurrency, and increasing likelihood of soft errors, the ability to rapidly identify and isolate errors will be essential. The total lack of a fault isolation strategy in modern OS's is arguably one of their weakest points moving forward.

- Old Conventional Wisdom (CW): CPU failure on an SMP node will result in a “kernel panic” that takes down the system. Such events will happen with increasing frequency in future silicon.
- New CW: CPU failure should result in isolation of the partition containing the failure. It would be better yet if the hardware supported integrated rollback mechanisms to support partition Restart. There is existing work in the Singularity OS to consider how transactions between the application and device interfaces can be rolled-back to a known state to support restart of partitions containing the application.
- Research Examples: VMM containers, Singularity.

6.1.3 Parallelism Scalability Challenges in Operating Systems

its role as the gate-keeper to shared resources, operating systems have traditionally been a major bottleneck in achieving scalability on SMP's. This is especially true for the open source Linux operating system, which has historically lagged behind commercial Unix OS's such as AIX and Solaris in scalability but has now become the dominant OS of choice for high-end systems. Significant attention has been devoted by the Linux community over multiple years to bridge the scalability gap with commercial OS's, starting with efforts such as improvements to the Linux scheduler in 2001 [88]. More recent examples of scalability efforts explored and undertaken by the Linux community include large-page support, NUMA support [95], and the Read-Copy Update (RCU) API [101]. While these Linux enhancements have resulted in improvements for commercial workloads with independent requests and flow-level parallelism [140] on small-scale SMP's, the scalability requirements for even a single socket of an Extreme Scale system will be two orders of magnitude higher than what can be supported by Linux today. It is clear that this gap cannot be bridged by business-as-usual efforts; in fact, future scalability improvements in Linux are expected to be harder rather than easier to achieve, as evidenced by the RCU experience [101] and the complexities uncovered by ongoing efforts to reduce the scope of the Linux Big Kernel Lock (BKL) *e.g.*, see [114].

High-end systems typically use specialized operating systems for compute and I/O nodes, and standard operating systems for service, front-end, and file-server nodes. In the case of Blue Gene/L [105], the specialized OS operates at the level of a *processing set* (pset) which consists of one I/O node and a collection of compute nodes. The design of the Compute Node Kernel (CNK) was simplified by placing a number of restrictions on the application *e.g.*, single thread per processor, absence of virtual paging, and support for only 68 system calls in Linux. While this design approach was necessary to support the schedule and system requirements of the early Blue Gene/L systems, it will not be practical for Extreme Scale systems with a thousand cores per chip or for the dynamic, asynchronous, and irregular parallelism structures expected in future grand challenge applications for Extreme Scale systems (Chapter 4).

September 14, 2009

Page 56

ECSS Report

The scalability challenges in operating systems of course extends to parallel file systems as well. In recent years, a significant amount of research on parallel file systems has been reported, including Lustre [99], GPFS [70], PVFS [96], pNFS [80], PanFS [107] and others [41, 52, 108, 112, 127, 143, 151]. Different APIs to interface with files such as MPI-IO [106], HDF5 [77] and NetCDF [115] have gained popularity as an alternative to the basic POSIX API. Object-Based Storage [133] provides a different way to organize data and metadata on the storage medium than file or block methods. Much of the increased functionality of these parallel file systems comes at the cost of increased complexity and overhead of the file system software. Some recent work seeks to reduce overhead of the file system and its load on the file servers. The Light Weight File System (LWFS) [111], a current project at Sandia National Laboratory, is a parallel file system that implements only essential functionality without any additional functionality that degrades performance and scalability. Implementations of additional features are moved into libraries and the application itself, allowing the application to be optimized to a right-weight solution.

6.2 Runtime Challenges

Runtime support for parallel programming requires key innovations in lightweight mechanisms for communication and memory hierarchy management, and user-controllable policies for managing the system resources. Expected contributions to this area of research include:

- Lightweight runtime mechanisms to exploit the novel features of interconnection networks, including topology queries, atomic operations, remote procedure invocation, fast one-sided transfer notification used in synchronization.
- Extensions of the execution models to handle fast and slow memory associated with a single thread, and demonstration of that model on a single-chip system with software-managed local memory that replaces or augments the traditional hardware-managed cache hierarchy.
- Runtime support to virtualize the set of processors through the use of multi-threading and dynamic task migration. Programming model extensions that allow for such virtualization when needed, without enforcing it for all applications.
- Runtime support for memory system virtualization, including object caching and migration. As with processor resources, the programming model will be extended to permit runtime-managed data placement in addition to the user-managed placement already available.
- Support for multiple runtime systems for different execution models and soft real-time applications.

6.2.1 Task Scheduling and Locality Runtime Challenges

Past task scheduling runtime systems have typically been optimized for dynamic parallelism that is oblivious of locality (*e.g.*, Cilk, OpenMP, Intel Thread Building Blocks) or for locality in the absence of dynamic parallelism (*e.g.*, MPI, UPC, CAF). As discussed in Chapter 5, a desirable characteristic for future programming models is that they express large amounts of concurrency with locality control so as to be “forward scalable” to future generations of parallel hardware. However, efficient locality-sensitive scheduling of $O(10^{11})$ lightweight tasks is a major research challenge.

6.2.2 Communication Runtime Challenges

High performance computing (HPC) systems implementing *fully-connected networks* (FCNs) such as fat-trees and crossbars have proven popular due to their excellent bisection bandwidth and ease of application mapping for arbitrary communication topologies. However, as extreme scale systems move towards millions (and even billions) of processors, FCNs quickly become infeasibly expensive. These trends have renewed interest in networks with a lower topological degree, such as mesh and torus interconnects (like those used in the IBM BlueGene and Cray XT series), whose costs rise linearly with system scale. Future systems will also need to take into account wiring complexity as they consider alternative low-degree interconnect topologies.

Indeed, the number of systems using lower degree interconnects such as the BG/L and Cray Torus interconnects has increased from 6 systems in the November 2004 list to 28 systems in the more recent Top500 list of June 2008. However, it is unclear what portion of scientific computations have communication patterns that can be efficiently embedded onto these types of networks without advanced runtime support to more efficiently map the communication graph onto the underlying hardware interconnection topology.

Figure 6.1 shows the communication patterns of a broad-variety of applications at a modest parallelism. Even at 256p concurrency, the analysis highlights the application's irregular communication patterns, evincing the limitations of 3D mesh interconnects [123]. At the same time, most communication patterns are sparse, revealing that the large bandwidth of a FCN is not necessary and have good locality, showing that an intelligent task-to-processor assignment can significantly decrease the load on the network. It is clear that either the interconnect will need to adapt to the diverse communication requirements of the applications, or there needs to be a system to implement static task placement or runtime migration of tasks to better map the communication topology onto the underlying topology of the hardware. The runtime system will play a crucial role in either case.

Current programming practice presumes an entirely flat model for communication locality, where every processor is equidistant to its peers. Although topological hints exist in MPI, they are rarely, if ever used. Most PGAS programming models only express two-levels of locality — local and remote. HPCS languages such as Chapel and X10 attempt to mitigate this by allowing the programmer to express locality using “locales” and “places”. However, optimized mapping of places onto hardware elements by the runtime system is still a major open problem.

There must be substantial changes in software practice to better expose communication requirements. If explicit task placement is left to the application developers, then performance portability may be brittle. This further supports the idea that the runtime system will need to play a more important role in task placement and interconnect configuration in future systems.

6.2.3 Synchronization Runtime Challenges

Applications that need to exploit high levels of hardware parallelism are usually expressed in terms of deep software parallelism and perform a significant amount of synchronization operations at various levels of the system hierarchy. Fast synchronization primitives might define several areas of research.

- Intra-node synchronization and notification: Synchronization primitives are usually implemented using either signals/interrupts or polling. Polling is the faster technique since it does not require any context switches but it makes implementations that have to deal with unexpected events cumbersome. One generic question that needs to be addressed in any implementation is the servicing of notification events. In some cases, these notification events initiate complicated execution paths that consume processor resources. As an example consider the

Exascale Software Study

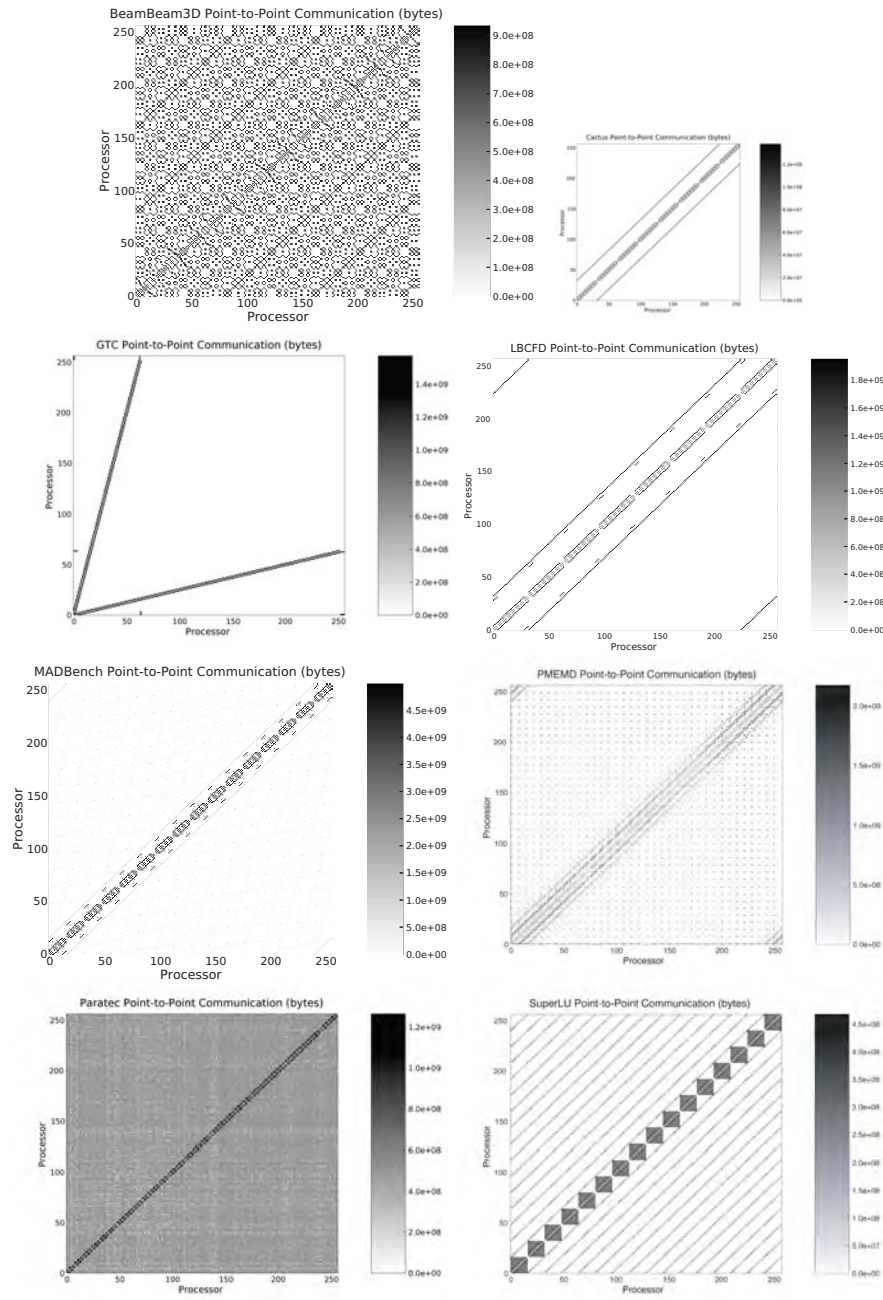


Figure 6.1: Topological connectivity of a broad range of scientific computing applications, showing volume of communication at $P=256$.

asynchronous remote execution functionality proposed in the DARPA HPCS languages. In the ideal case, execution of unexpected events is dispatched to the “execution unit” that already has possession of the required resources. Examples are requests for data that already resides in a processors cache or requests for execution of services that a processor has just served. One research direction will be interrupt dispatch based on resource requirements.

- Remote synchronization: The networking layer for advanced execution models supports several primitives for remote synchronization. Depending on the hardware target, these primitives are implemented using either active messages [142] or specific system features (such as offload). We envision a highly concurrent and asynchronous execution environment where events are generated and dispatched to various system components with various resource allocations and privileges. In this setting, asynchronous events are ideally dispatched to the entity that “expects” them and also has the available resources. The necessity for a distinction between resource ownership and availability to serve an event is illustrated by networking layer implementations of data packing/unpacking communication primitives using AMs. Whenever an active message requesting data packing arrives at a node, it should ideally be served at the processor that's most likely to have the data in caches. However, if that processor is busy, the request should be served by a different core.

Several requirements become apparent from the previous discussion. Events have to express their resource requirements, execution entities (threads) have to be paired with resource usage (*e.g.*, memory footprint) and schedulers/dispatchers have to be able to access this information. Meeting these requirements translates into research into performance instrumentation and techniques to extract and describe application behavior.

6.2.3.1 Thread and Resource Virtualization

Current threading packages, such as Pthreads, offer very limited control over scheduling decisions. Ideally, information about resource utilization is associated with threads and schedulers take this information into account. Furthermore, experience indicates that cooperative threading rather than preemption is able to increase performance in a HPC environment. This unfolds into several research directions:

- Develop mechanisms to extract resource usage (memory, functional units) for parallel applications, using both dynamic (instrumentation and runtime monitoring) and static (program analysis) approaches.
- Develop mechanism to extract the current state of execution: precise point in the execution thread and expectation of near future action.
- Develop scheduling mechanisms and policies based on resource usage, priority and data dependence information.
- Develop mechanisms and policies to avoid deadlock in a cooperative scheduling environment.
- Develop frameworks to expose this functionality to applications/libraries.
- Develop mechanisms to ensure progress and attentiveness in the presence of asynchronous remote execution and Active Messages based implementations.

6.3 Compiler Challenges

The crucial role of compilers at the extreme scale is to map from language constructs that express a very high-level decomposition of an application to highly power-efficient and memory-efficient architecture-specific code and runtime layer calls. As has been proven historically, completely automatic compiler optimization from high level code will not meet the performance requirements at the extreme scale; in the extreme scale regime, we will also encounter memory and power constraints that programmers and tools could previously ignore. Further, compiler-based approaches, such as, for example, compilers for PGAS languages, have generally focused on regular, static parallelism. As the applications for extreme scale platforms expand to encompass irregular, unstructured and dynamic algorithms, so must the compiler technologies that support these challenging application domains. An article reporting on a recent NSF-sponsored workshop on the future of compiler research listed the following 6 research challenges, in addition to other guidelines on enhancing research and enriching education [75].

Compiler research challenges in optimization include:

- Make parallel programming mainstream;
- Write compilers capable of self improvement (*i.e.*, auto-tuning); and
- Develop performance models to support optimizations for parallel code.

Compiler research challenges in correctness include:

- Enable development of software as reliable as an airplane;
- Enable system software that is secure at all levels; and
- Verify the entire software stack.

Compilers at the extreme scale must collaborate closely with the application programmer to derive an architecture-independent algorithm description that can be mapped to high-quality code; further, the compiler must incorporate lightweight mechanisms that interface with the runtime layer and architecture to dynamically map this code for a specific execution context to be both high performing and power efficient.

6.3.1 Customized and Dynamic Code Generation

Generally speaking, the role of code generation must fundamentally change in response to the need for agile code mappings that respond dynamically to execution context, including input data set properties, machine load and power constraints. Rather than a single statically-generated implementation of a computation, the compiler must represent a space of possible implementations that are generated either a priori, by predicting relevant features of execution contexts, or dynamically, in response to execution context.

Some of the critical decisions made in the code generation process must be exposed to both programmers and the runtime layer, and the set of alternative implementations must be well-defined and systematic. Such requirements will make it possible to explain the code generation process to the application programmer, and to mechanically evaluate these alternatives, either off-line or dynamically during execution. An essential feature of code generation is that the mechanisms, either dynamic code generation, partial code generation that is instantiated at run time, or run-time selection of statically-generated code, be efficient in both execution time and memory requirements.

6.3.2 Extracting Useful Parallelism from Ideal Parallelism

At the language level, as discussed in the previous chapter, we want the application programmer to be able to express an abundance of parallelism, providing the underlying system architecture with sufficient degrees of freedom to derive an efficient mapping of the parallelism. We need to strike a delicate balance between runtime overhead for managing parallelism and the risk of idle resources or load imbalance from lower overhead static thread management.

Historically, compiler-managed parallelism is mostly static, and possibly explicitly declared by the programmer. In the extreme scale regime, the compiler should make some decisions on static management of parallelism for efficiency's sake, and defer other decisions for the runtime layer. Therefore, the virtualization of some but not all parallelism is desirable, and even for runtime decisions, the compiler should optimize for efficiency of dynamically-mapped code. The programming model can assist with how to decide what parallelism to bind statically, and what to defer to runtime, and the complexity of runtime decisions. For example, a hierarchical expression of parallelism would provide the compiler with useful guidance on how to do this mapping [116, 117].

6.3.3 Optimizations for Vertical and Horizontal Locality

A wealth of prior research on compiler optimizations to manage locality for uni-processors will provide an important foundation for both vertical and narrowly-defined horizontal locality within a chip. Dramatic reductions in memory per core and increased competition for off-chip memory bandwidth will make such optimizations truly essential, but the approaches taken must be modified to support parallel threads and in particular sharing of data across threads. Where caches are shared across cores, fine-grain scheduling of parallelism to exploit locality in aggregate caches will be needed. Further, on-chip non-uniform access time to caches (*i.e.*, NUCA architectures) may require careful placement of data even in caches.

Across processor chips and across the storage and processor hierarchy, careful data layout in memory will be required to optimize performance and manage power. While PGAS and HPCS languages offer some support for expressing data layout, as discussed in the previous chapter, new programming model constructs are needed to express hierarchy and manage locality in light of dynamic parallelism. Further, user-defined data layouts and libraries, working in conjunction with compiler-generated code, will be required for more irregular applications. The role of the compiler is to map high-level data layouts into levels of the memory hierarchy.

6.3.4 Synchronization and Communication Optimizations

Extreme Scale environments will demand advanced compiler analyses and optimizations that expand on current communication optimizations. Such optimizations must extend the scope of compilers to handle (or even specify) new runtime mechanisms for synchronization, caching, and other critical dynamic decisions based on user-space scheduling, memory management or communication.

6.3.5 Energy Optimizations

Compiler optimizations to reduce energy consumption have been a topic of research interest for over a decade, but have mostly been deployed only in embedded environments. In conventional architectures and high-end systems alike, energy is largely managed directly by hardware and low-level system software. It is still the case that most compiler optimization research has focused on minimizing execution time, without regard to power. For extreme scale architectures, the compiler's optimization objective function must consider both performance and power. Fortunately,

some optimizations can be good for both objectives, such as increasing processor utilization or improving data locality. However, improving performance and managing power might be at odds with each other in parallelization since a faster time to solution may use resources less efficiently in computations that do not exhibit perfect strong scaling. Therefore, situations arise in which the compiler must balance performance and power to obtain a solution that meets both sets of objectives, suggesting the need for incorporating estimates of power consumption and power budgets into the optimization process.

6.3.6 Support for Resilience

As discussed in [75], future requirements call for continuing the trend of the increasing role of compilers in detecting errors automatically or semi-automatically. Research must continue to be pursued in static and dynamic analyzes, in conjunction with language constructs, to detect programmer errors. Further, the compiler will be responsible for appropriate code generation mechanisms to detect hardware and system software errors and respond to faults.

6.3.7 Global Auto-tuning and Dynamic Optimizations

Compilers will play an important role in auto-tuning and dynamic optimizations, systematically applying the set of optimizations described in this section to empirically evaluate the best-performing solution. Auto-tuning could be used for managing both performance and power, for computation kernels and whole applications, and in both off-line and on-line settings. Because auto-tuning as a technique requires a number of technologies that are beyond the capabilities of a compiler such as techniques for navigating prohibitively large search spaces, library, run-time and application-level optimizations, further discussion of auto-tuning is deferred until Chapter 7.

Dynamic optimization strategies must be efficient enough to employ at run time, or must be used in conjunction with partial code generation and dynamic instantiation or a priori code generation and dynamic code selection. The overhead of dynamic optimization means that it must be applied at the appropriate computation granularity. If high speedup is feasible for a repeated computation, then the execution can also overlap optimization with execution of a previous time step and execute the newly optimized code when it becomes available.

6.4 Library Challenges

One of the many tenets of computer science is an aspiration to develop techniques to manage and reduce software complexity. Often the practical limitations on complex computer systems are human comprehension — not the physical computing and I/O capacity. This is especially true of high-end computing (HEC), with a heightened emphasis as we scale to petascale and on to exascale systems. Earlier in Section 4.8, we discussed the role of application frameworks in reducing software complexity. In this section, we discuss the role of *software libraries*.

Libraries have historically been an important means to managing the complexity of system software. In the HPC community, libraries can provide both productivity and performance. The domain-specific libraries discussed earlier in Section 4.8.5 make use of support libraries to provide optimized performance across a wide range of architectures – possibly ranging from workstations to Top10 systems. The relationships between domain-specific and support libraries are depicted in Figure 6.2. Note that support libraries may be architecture dependent with (1) numerical libraries optimized for specific microprocessors, (2) communications libraries optimized for specific system interprocessor communications characteristics, and (3) data management libraries and I/O

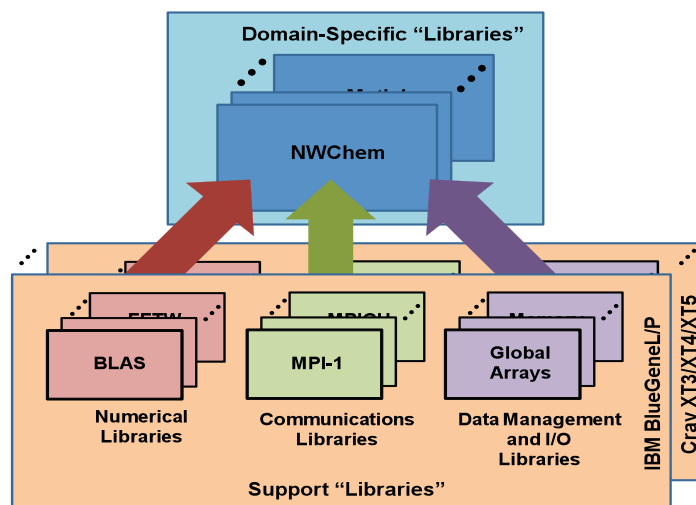


Figure 6.2: The Relationships between Domain-specific and Support Libraries

libraries optimized for specific shared memory and I/O system characteristics. It is important to note that support libraries — numerical, communications, and data management — can be used by any program and need not exclusively be used in a hierarchy below domain-specific libraries and frameworks.

6.4.1 Numerical Libraries

Numerical libraries have been available to support application developers for over 40 years. The International Mathematics and Statistics Library (IMSL) may be one of the oldest general numerical libraries, currently supporting a comprehensive set of 1000+ algorithms [16]. The Basic Linear Algebra Subprograms (BLAS) was first published in 1979 and is used to develop other numerical libraries such as LAPACK [17] [18] [19]. An important feature of numerical libraries is that they offer both portability and performance. While open source software is available to compile and link for any architecture, the highest performance most often comes from highly optimized libraries for specific processor/system architectures. To minimize the effort to reach high performance, an open source auto-tuning implementation of BLAS APIs was developed, notably ATLAS (the Automatically Tuned Linear Algebra Software) [20].

The algorithms in numerical libraries have evolved as a function of architecture over time. This is illustrated in figure 6.3. LINPACK in the 1970's was based on level-1 vector-vector operations, which were well matched with the vector architectures of the time. LAPACK in the 1980's needed to deal with caches, data movement, and locality and used data-reuse friendly methods that employed level-3 BLAS routines. With the advent of massively parallel processing with distributed memory in the 1990's, LAPACK evolved into ScaLAPACK based on PBLAS message passing. The most recent incarnation of numerical libraries must be able to work on new many/multi-core architectures where thread-level parallelism is as important as distributed memory parallelism.

The Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) project aims to address the critical and highly disruptive situation that has been a result of the introduction of many/multi-core processor architectures. Like the BLAS, LINPACK, LAPACK, and ScaLAPACK,

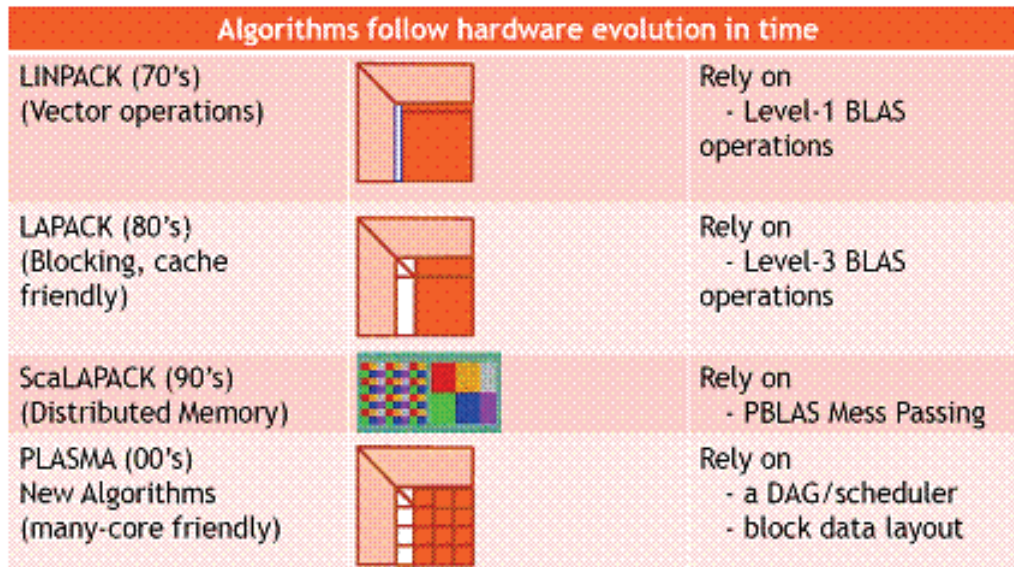


Figure 6.3: Figure Evolution of Optimized Algorithms in Numerical Libraries [55]

PLASMA's ultimate goal is to create software frameworks that enable programmers to simplify the process of developing applications that can achieve both high performance and portability across a range of new architectures. The new technologies employed in PLASMA are based on asynchronous, out of order scheduling of operations as the basis for the definition of a scalable yet highly efficient software framework for linear algebra applications [21]. Other open source numerical libraries include Fast Fourier transforms (FFTs) such as FFTW [22], FFTE [23], and Vector Signal Image Processing Libraries such as VSIPL/VSIPL++ [24].

Nearly all open source numerical libraries have been designed to deal with issues relating to portability and performance. Many numerical libraries offer simple portability with reference implementations that can be compiled and linked with larger applications. Many of the numerical libraries offer both portability and performance. Some of the aforementioned libraries have automatic tuning features *e.g.*, FFTW. Others have been optimized for particular architectures *e.g.*, FFTC, which has been optimized for the Cell BE processor architecture. Others rely on either hierarchically employing optimized vendor libraries or having vendors develop optimized implementations *e.g.*, VSIPL/VSIPL++.

While the HPC community makes extensive use of open source numerical libraries, there are optimized numerical libraries available provided by vendors *e.g.*, Intel and IBM both provide optimized numerical libraries for processors they have developed. Intel provides the Math Kernel Library v10.x (multi-threaded and thread-safe) [25]. IBM provides the Engineering Scientific Subroutine library (ESSL) (thread-safe) and the Parallel ESSL (based on MPI for distributed memory applications) [26]. These libraries are often available free for non-commercial use.

6.4.2 Communication Libraries

Communications libraries come in two basic varieties — general purpose and domain-specific — with the possibility that domain-specific communications libraries have been built upon general purpose

communications libraries. We traditionally think of “libraries” as using language subroutine or function calls, but here we permit the more general idea of an application programming interface (API). MPI (and its variants) is based on the subroutine/function model [27]. In the past, there were application specific communications toolkits *e.g.*, TCGMSG that was provided with the Global Arrays support software that is part of NWChem [28]. Later versions were built upon MPI (*i.e.*, TCGMSG-MPI) but maintained the domain-specific flavor in the programming interface [29].

6.4.3 Data Management and I/O Libraries

The third support library category describes those libraries developed to support data management and I/O. There are general purpose libraries *e.g.*, MPI-I/O, which combines the portability and “look and feel” of MPI with performance and file interoperability [30]. Additionally, there are domain-specific libraries to simplify data management and I/O. For example, NWChem makes use of the following data management and I/O libraries:

- Global Arrays provides an efficient and portable “shared-memory” programming interface for distributed-memory computers [13].
- Dynamic Memory Allocator provides a dynamic memory allocator for use by C, Fortran, or mixed-language applications [31].
- Aggregate Remote Memory Copy (ARMCI) library provides a portable remote memory access (RMA) operations (one-sided communication) optimized for contiguous and noncontiguous (strided, scatter/gather, I/O vector) data transfers [32].
- ChemIO provides a standard I/O API that meets chemistry requirements while being portable and being highly efficient [33].

Chapter 7

Challenges in Supporting Extreme Scale Tools

Extreme Scale tools encompass the portion of the software environment that support human clients or other parts of the software stack. These tools must interact with producers and consumers of information in two directions, both to provide guidance and query for additional information.

- *Explaining application and system behavior:* Tools to collect and present information to the user or other parts of the software stack to help assist in the process of changing the application to improve its behavior.
- *Exploiting information beyond application code:* Tools to collect and utilize information from the user or other parts of the software stack to automatically improve the application.

We group all such tools collectively into *development environments*, which support the mapping of application code to specific architectures. Several broad categories of development environment capability include (1) *performance and power optimization tools*, which exploit architectural features and eliminate bottlenecks; (2) *correctness tools*, which pinpoint and eliminate errors and vulnerabilities; (3) *analysis of computation*, which provide analysis of application behavior, preferably through interactive visualization; (4) *application completion tools* to manage the low-level details of mapping an application to an architectural platform; and, (5) *compilers*, which provide support for all of the above, in addition to translation from the programming model. While more details on these classes of tools are presented in Appendix A.2, this chapter examines the technological challenges facing tool developers.

7.1 History of Tools and Development Environments

For well over twenty years, researchers and tool developers in academia and industry have been struggling to develop technologies and tools for tuning performance and debugging parallel systems and applications, with only partial success. The reasons for these struggles are manifold, but can be traced to a combination of technology, economics and human psychology.

Technologically, finding and re-mediating performance or correctness problems is difficult, given the complexity and scale of today's parallel systems. Not only do tens to hundreds of thousands of hardware components (processors, cache and memory systems, communication interfaces and interconnection networks, and storage systems) interact in oft-unexpected ways, their behavior is mediated by complex, multilevel system and application software hierarchies. Subtle interactions

among the components at any level, or even multiple levels, can adversely affect observed application performance. Each new, large-scale system exposes examples of such unexpected problems, from the deleterious, system-wide effects of a TLB replacement algorithm, to operating systems jitter induced by system daemons, to adaptive runtime systems that conflict with system policies. If history is any guide, such challenges will only grow with future extreme scale systems.

Despite, or perhaps given this complexity, software performance and debugging tools are rarely a priority during development for HPC system designers or vendors. Perhaps paradoxically, the shift to commodity-based HPC systems has exacerbated this difficulty, for software performance and debugging tools are one of the few aspects that are not readily extensible from the sequential domain. One can construct large-scale systems using entirely commodity components - processors, memory and storage systems, interconnect, operating system, libraries and compilers, yet the challenges of tuning and debugging one hundred thousand concurrent threads differ markedly from PC-based debugging and tuning.

Moreover, experience has shown that there is no economic market to stimulate development of parallel software tools. There is no thriving ISV market for HPC debuggers or performance tools, nor is development of such tools a priority for the large system vendors. The selection criteria for HPC system procurement overwhelmingly focus on other aspects — peak and sustained performance, reliability and power consumption — but less on human productivity and total time to solution. Government contracts mandate basic tools as part of most, if not all procurements, but this has done little to advance either the technical or commercial state of the art.

Finally, unlike other elements of the HPC software stack, software performance and debugging tools must combine both technology and usability. However arcane the compiler and code development interface, no current user would consider writing assembly code. In contrast, performance and debugging tools must be capable of capturing and presenting data on possible performance or correctness problems, and they must do so in ways that users find intuitive and helpful, else the users will eschew these tools in favor of more rudimentary alternatives such as manual instrumentation.

Given the technological complexity of tools, the realization that market forces seem unlikely to solve current problems, and the usability challenges inherent in tool development, several workshops and reports have recommended that we change our current research and development model. Simply put, our current tool approaches are not working and have not worked for the past twenty years. The remainder of this chapter examines how tool research can be integrated with research on extreme scale systems to navigate the complexities of $O(10^{11})$ -way parallelism.

7.2 Overview of Extreme Scale Development Environment Challenges

Many of the most critical challenges facing tools and development environments are variations on challenges that have plagued the HPC community for 20 years, as discussed in Section 7.1. Progress in this area demands a very different approach to developing and deploying tools. Beyond existing challenges, an extreme scale regime will dramatically increase the complexity of developing, debugging, modifying, porting and validating future applications. Hundreds of thousands of threads executing in an increasingly dynamic environment is beyond the intellectual scope of even the most veteran HPC application developers. The application programmer must of necessity let go of total control of performance and focus on using more productive ways of expressing and mapping their applications. Further, managing power consumption will become a new priority for application programmers. This growth in complexity across a number of dimensions makes the use of tools and development environments *even more essential* in managing the workload of HPC developers

and broadening the pool of talent for developing applications at this scale. We now enumerate the essential properties of extreme scale tools and development environments, and then discuss specific details of how these properties impact the technological direction of the extreme scale software stack.

Performability. One immediate consequence of the dramatic growth in system scale is the increased importance of system reliability and the need for alternate approaches to system resilience. Today's HPC systems contain more addressable nodes than the entire Internet did just a few years ago, yet our modus operandi continues to presume systemic reliability. With millions to tens of millions of hardware components, component failures will be frequent events, yet they should not trigger system failure. This suggests that future extreme scale systems must adroitly support a continuum of operating modes, ranging from complete health (all components operating correctly) to substantially degraded operation (many failures). From the tools perspective, this notion of performability (*i.e.*, integrated performance and reliability) means combining multiple techniques for fault tolerance (*e.g.*, checkpointing, redundant computation, restart-retry) with real-time monitoring to detect aperiodic component failures.

Scalability. When debugging and tuning a parallel application on a small-scale SMP, one has the luxury to capture detailed data on the fine-grained interactions among threads and hardware components. At the exascale, such detailed examination is neither productive nor even possible. The volume of performance and debugging data and the more worrisome perturbations induced by its capture become unmanageable. This suggests that extreme scale performance analysis and debugging tools must combine dynamic instrumentation techniques (*i.e.*, those that can be enabled and disabled rapidly and on demand) with methods that exploit large scale as an advantage. Stratified population sampling, adaptive compression, temporal logic and classification mechanisms can be used to capture data and reason about behavioral equivalence classes. Only with such approaches can instrumentation infrastructure and overhead grow sublinearly with system size.

Abstraction. The usability counterpoint to scalability is abstraction — presenting system and application behavior in terms relevant to the system operator or application developer. In turn, this has implications for program transformations and compile-time and run-time optimizations, as crucial information must be preserved across transformation boundaries, else there will be inadequate data to relate measured behavior to application specifications. Moreover, if we are to broaden the base of HPC application developers, we must move to presentation metaphors that are deeply tied to the application model, not the hardware. Although extreme scale software developers, as with any apex system, will be willing to accept the need to understand some level of system detail, this will be unacceptable to users of departmental extreme scale systems. Knowledge of multicore chip structure and interconnects, transient bit errors, memory hierarchies and interconnect topologies should not be required to develop portable, reliable, efficient extreme scale applications. This is especially true when a level of hardware-software virtualization will likely be required to hide ongoing component failures.

Adaptation and Autotuning. As noted, the volume and complexity of performance and debugging data are likely to overwhelm even the most determined application developer, even when abstracted and presented in terms relevant to the application programming model. This suggests that manual optimization should be complemented by dynamic adaptation and autotuning. Such introspective runtimes would combine real-time measurement, via targeted sensors, decision procedures for intelligent policy configuration and selection and actuators for decision implementation. Implementation challenges include hysteresis (*i.e.*, the lag between change and its manifestation), oscillation (*i.e.*, avoiding repeated policy or configuration changes) and multilevel adaptation, where compiler-synthesized adaptation (*e.g.*, multiversion code generation), runtime library adaptation (*e.g.*, scheduling or code dispatching) and operating system management (*e.g.*, virtualization and

dynamic provisioning) interact appropriately and intelligently.

Multilevel Integration As noted earlier, one of the many challenges of complex HPC systems is the behavioral interdependence across hardware and software levels. As we raise the level of programming abstraction to increase human productivity and to hide the idiosyncrasies of specific hardware implementations, the semantic gap between user specifications and run-time behavior will continue to widen. Data parallel languages, functional specifications, domain-specific toolkits and libraries all hide system details, and their implementations depend on multilevel translation and mapping. Understanding the performance and assuring the reliability and correctness of applications written using these tools will only be possible if the hardware and software tool chain contains information sharing specifications and interfaces that can relate measured data to application specifications.

Availability and Portability. When tools are provided by vendors of specific platforms, application developers must either focus on using only a set of platforms sold by that vendor, or use different tools as they port from one platform to another. Neither of these strategies is reasonable, and undoubtedly has contributed to the general lack of adoption of tools as part of the application developer's workflow. An alternative and more desirable solution is the existence of robust tools that are available on every HPC platform, preferably open-source tools that can be adopted in academic environments and taught to the next generation of application developers. The motivation to develop and deploy such tools must be carefully examined, with appropriate incentives to guarantee a long-term evolution and maintenance of portable, robust and low-cost tool and development environments. Fundamentally, the path to robust, efficient and effective tools and programming environments demands that such software must be an integral part of an overall system design, and not as an afterthought when the system has already been developed.

7.3 Enabling Technologies for Exascale Tools

Given the previously-described requirements, as part of this study we have identified three key enabling technologies that will be essential for tools in exascale systems. The first of these is the ability to collect and analyze enormous volumes of data to improve an application's execution. The second considers the computation side of data collection and analysis: how to enlist additional computation to improve the execution of the main application. Finally, we discuss *autotuning*, a principled methodology for using additional computation to test out alternative mappings of a computation to find the best implementation.

7.3.1 Scalable Data Collection and Analysis

Data Collection. While almost every microprocessor provides performance counters to examine execution data, these counters fall short of the requirements for exascale tools. Fundamentally, hardware performance counters collect low-level information that is not directly meaningful to application programmers. Vendors find these counters useful in understanding performance bottlenecks on existing workloads, and improving upon functionality in future generations of devices. Deriving meaningful application performance data from such low-level counters requires the programmer or tool developer to interpret the meaning of the counters, multiplex the counters of interest since only a small number of events can be counted, modify the application to access the counters, and then interpret the results and decide how to modify the application. Further, there are gaps in what the counters are collecting, such as, for example, communication events from the interconnect.

Software layers can also inhibit data collection. The goal of exposing performance counter information in an architecture-independent way, while desirable, is somewhat elusive. Portability issues are a frequent challenge when counters on different platforms are not counting the same thing. Compilers also introduce a number of challenges in mapping collected data back to code structures in the application code. Following lowering from high-level constructs and optimization, the compiled code may look very different from the original application. Only with proper preservation of the original structure during the compilation process is it possible to do this mapping [135,136]. Run-time layers and operating systems face similar challenges in mapping dynamically collected data back to application constructs. A further requirement for dynamically collecting data in software is its overhead, since a subset of data collection may need to occur during production executions. The importance of efficient data collection in exascale systems begs the question of whether the design of hardware counters, compilers, operating systems and development environments might be very different if the designers start with the goal of explaining application behavior.

Autonomous Data Analysis and Mining. The system must autonomously and efficiently collect data about what the program and system are doing, and perform on-the fly analysis of the result at scale. Due to the complexity of exascale application behavior, the system must find anomalies that are unanticipated, and therefore must collect data on routine and production runs, and not just when looking for a specific problem. Thus, efficient modes of collection and analysis become a high priority, as compared to detailed traces and post mortem analysis.

On-the-fly analysis algorithms must be developed that rapidly reason about behavior and compare it to expectations, perhaps with system support to maintain efficiency. What are the baselines or ground truth to which an execution can be compared? As an example, in monitoring SPMD programs, analysis could look for threads with the most deviant behavior. Other baselines could include user-specified expectations, models of expected behavior or results of prior execution. In support of automatic validation of applications, scalable techniques for off-line detection of common errors must be developed.

Implicit in this analysis is a set of important decisions about what information to collect, analyze and discard or retain. The retained data must be organized in a performance database that provides meaningful information to subsequent executions or phases in the current execution, but also provides efficient access for on-the-fly analysis. Further, it should be noted that while there may be some differences, much of the underlying support for identifying performance, power or correctness anomalies will be common.

7.3.2 Companion Computations

While some applications may achieve high utilization on Extreme Scale systems and remain compute-bound for their entire duration, many applications will likely be limited by other aspects of the system such as memory bandwidth for at least some part of their execution. For these applications, the unused hardware resources can be used to assist in improving critical application characteristics such as programmability and resiliency. The notion of utilizing otherwise wasted resources to improve execution is not a new idea, and has been proposed to exploit unused issue slots for ILP, in multithreaded architectures, and now for multiple cores. An available surplus of resources suggests mechanisms to support additional threads or processes not directly contributing to computation, which we call *companion computations*. A companion computation provides information about execution of the main process that can be used to analyze or improve performance, identify the health of a node to improve reliability, examine tolerances to ensure accuracy, increase throughput and interface with developer or other tools. Companion computations can be both sensors, detecting problems during execution, and actuators, modifying execution to improve its behavior.

The notion would be to allow the user/tool to start up a companion computation at the same time that the application is started. The companion computation would not only have its own processors and memory, it could read and write into the memory of the executing application. The companion computation would also have access to the interconnect and be able to communicate with other companion computations associated with the same application.

The operating system should permit and assist in co-location of application and companion threads. In addition to the reading and writing of the application's data, the companion computation must be able to multiplex hardware counters that are important to the determination of the bottlenecks in the application. Monitoring message traffic is also extremely important.

On the process side we have the ability of the companion process to capture important information about the execution of the application and at the user/tool end we have the interface to allow the user to direct the function of the companion process.

In designing mechanisms and roles for companion computations, we must adhere to the premise that companion computations should do no harm. If the purpose of companion computations is to gain useful system throughput, it is problematic if they are competing for resources with the main computation.

7.3.3 Autotuning

Autotuning is broadly used to describe application code, libraries, and compiler-generated tools that, either in an off-line or dynamic way, evaluate a set of alternative implementations. This evaluation usually involves executing code under representative execution contexts, to select the most appropriate for a specific hardware platform, input data set, and execution environment. As with companion computations, off-line autotuning is feasible in today's powerful systems, and on-line autotuning will become feasible in light of the vast resources available in exascale systems.

The popularity of autotuning arose in response to the complexity of modeling or predicting the impact of code changes on performance, particularly given the subtle interactions between hardware features. This complexity will grow in the exascale regime, but so will the availability of hardware resources to be used in the autotuning process. At exascale, on-line parallel search of alternative implementations becomes more feasible.

While auto-tuners have largely focused on improving performance, the general approach is well-suited for other optimization criteria, such as reducing power consumption, increasing throughput, or limiting the load on some overused resource such as interconnect. Support for auto-tuners might include appropriate hardware performance counters, parallel search algorithms and heuristics to prune the search space of potential implementations, programming model features to express both the space of possible implementations (as discussed in Section 5.5) and ways to prune this space, and compiler technology to map the set of implementations to executable code segments.

7.4 Scenarios for Interaction with Tools

This section describes scenarios for how tools may interact with the exascale software stack to manage the complexity of application development, execution and understanding/modifying tasks. In each scenario, we present a motivation for why new approaches will be needed in an exascale regime, and connect the scenarios to the six requirements in Section 7.2 and the three enabling technologies in Section 7.3.

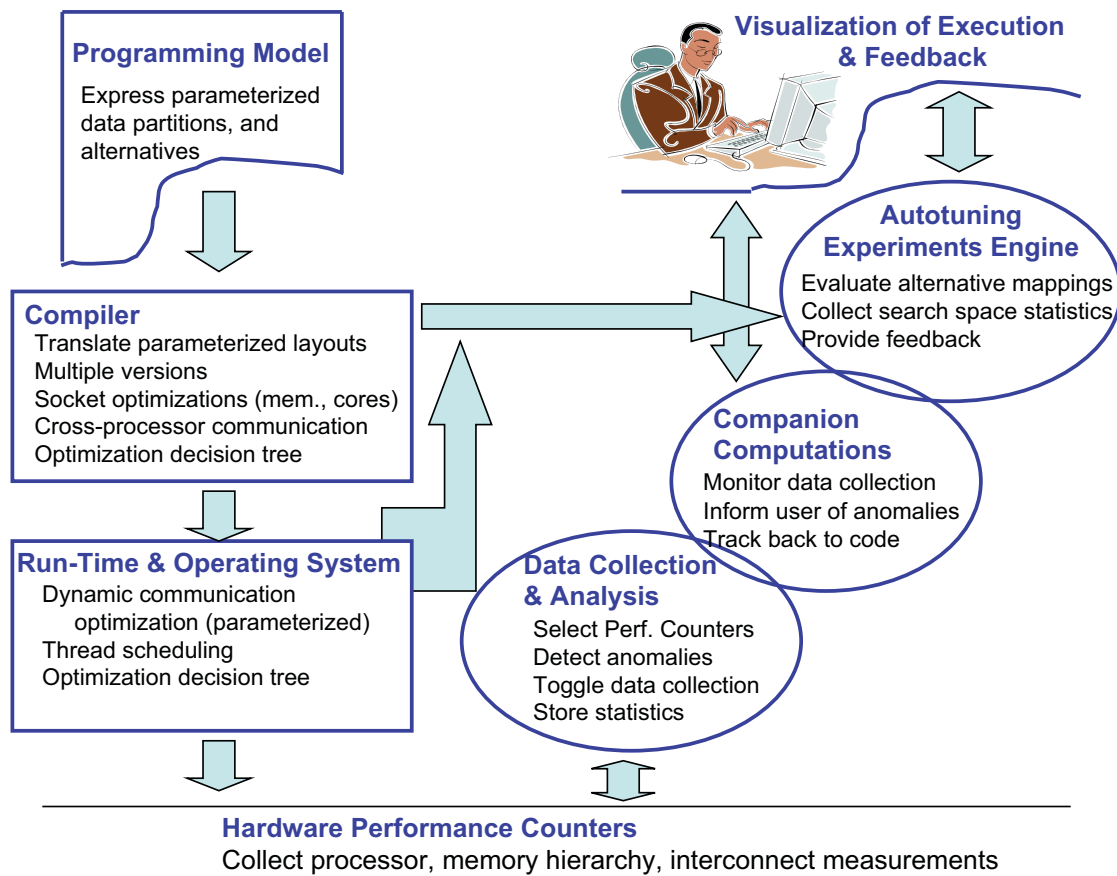


Figure 7.1: Illustration of information flow for Analyzing Data Partition

7.4.1 Scenario 1: Clarify Performance Behavior for Developer

Abstraction involves distilling relevant behavior so that it is meaningful to the developer and helps them implement changes that improve execution of their application. This requirement demands that the system pinpoint causes of problems and map them back to application constructs, and also provide the developer with suggestions of changes that have a direct and observable impact on execution behavior. Therefore, the previously-described multi-level integration must support, from initial design, mechanisms for such changes, and measurement capability to provide feedback on the effectiveness of such mechanisms.

To understand what the developer needs to know we look back to the early days of vector processors, when the execution behavior was easy to discover from feedback from the compiler and profiling tools. The developer simply had to go to the routine in the application where the time was being spent and then determine (with compiler assistance) if the code vectorized, and if not, why.

With the advent of large-scale and distributed-memory architectures, the reasons why the application is not performing well are more obscure and more difficult to remediate. When the developer specifies a potential decomposition approach (parameterized by problem and machine size) for their application data, this single choice has a significant impact upon the performance of the application. Once the decomposition is specified, a number of other performance features of the implementation are set as well, such as load balancing and number of messages. Additional optimizations on the message-passing and overlap with computation can further improve performance, including message consolidation and asynchronous pre-posting of receives. Once an application is in production use, the modification of the decomposition is typically very difficult and the optimization of the implementation of the message passing is the only way to improve the performance. Virtualization approaches as in CHARM++ [84] can ameliorate this problem by allowing the programmer to specify an “over-decomposition” and using the runtime system to perform dynamic load balancing.

Using a high-level language, for example High-Performance FORTRAN (HPF), the decomposition is easy to change; however, the optimization of the implementation is left to the compiler. Among the many lessons to be drawn from the HPF experiences of the 1990s is the need for invertible performance mappings that can relate the measured behavior of executing code to application programming idioms. HPF compilers generated message passing code (typically MPI) from data parallel FORTRAN, annotated with data distribution directives. Presenting message passing metrics, derived from instrumenting the generated code, to an HPF application developer was of no value, as the application developer had no way to either understand the performance problem or to change it. What was needed was a mapping from measured communication patterns to recommendations about changes to data distribution directives or code fragments.

The average application developer who uses the high level language would have little or no knowledge of the implications of parallelizing their application. Where there was a simple, “Did it Vectorize?” question on the vector machines, on the large scale parallel systems, the questions are significantly more difficult to ask. To address this challenge, the compiler/runtime system must be very precise and accurate as to the major reasons for the lack of scaling. As the level of abstraction rises, the difficulty of relating certain performance abnormalities to the actual code becomes much more difficult. The importance of solving this problem is at the root of the issue of using new languages that reduce the difficulty of using explicit message passing.

Thus, in Figure 7.1, we illustrate how the exascale software stack might collaborate to find an appropriate data layout for a computation. We rely on abstraction and information sharing, which have been among the most successful of our software ideas. Programming models and languages, libraries and runtime systems, operating systems and hardware, all define abstractions and

information sharing interfaces. However, these interfaces are predominantly lossy and communicate information in one direction — from above to below. To implement run-time adaptation and performability at exascale, we must increase the volume and types of bi-directional information sharing. For our performance optimization scenario, compilers must retain code transformation data to allow performance tools to relate measured data to application source code in ways that suggest how application code might be changed to increase performance or reliability.

Let us assume the programmer expresses in the programming model a parameterized data partition, with actual sizing of dimensions of the layout left to be bound empirically. The programming model may also allow specification of multiple organizations and partitions, *e.g.*, for a sparse graph, that can be compared. The compiler translates each partitioning strategy into parameterized code, possibly generating multiple alternatives that can be selected at run time. The compiler generates cross-processor communication, hierarchical parallelization across cores, and a set of optimizations to exploit the memory hierarchy. For each set of optimizations the compiler performs, it may have unbound parameters that can be set empirically. It may also have a set of decisions that it would prefer to defer until run time. Similarly, the run-time system will dynamically perform thread scheduling and optimize communication, and the run-time too may parameterize these optimizations or have a set of decisions it would

We can think of the result of compilation as a set of alternative mappings, rather than just one, and execution as a comparison of multiple different execution strategies. This suggests a complex evaluation process to arrive at the most appropriate mapping. An autotuning experiments engine evaluates the alternative mappings, providing feedback to the application programmer as to what mappings were most successful. To support this evaluation process, the hardware must provide access to performance counters to describe (among others): (1) processing within a core and within a socket; (2) memory hierarchy behavior; and, (3) from the on-chip and cross-socket interconnects. Companion computations derive important performance metrics and look for abnormal indicators that could indicate poor performance. On-line collection and analysis of data steers the search for the most appropriate mapping, with a tiny subset of results added to the performance database. For example, if a companion computation for data analysis observed load imbalance, it could initiate a tracing of messages coming into and going out of the process, combined with workload characteristics, to understand how the data layout is contributing to this performance problem. With such real-time access to the application, a companion computation could provide a sophisticated interactive visualization of the execution, providing the user multi-level access to performance across a set of experiments for different mappings. This interactive interface would be designed to correlate the application details to the location in the program responsible for the performance details. Through automatic evaluation of a range of implementations and online comparison across these implementations, such an approach would allow the programmer a view of how the layout specification impacts performance.

7.4.2 Scenario 2: Online Failure Detection and Response

Historically, we have treated performance and reliability as distinct objectives, with equally distinct approaches to design and optimization. For scientific and technical applications, checkpointing is the standard mechanism, where the nodes of an MPI application collect and write data to secondary storage, typically every few hours. These checkpoints can be used to restart the computation at a later time. Implicit in this approach is the assumption that failures can be detected reliably. Further, our performance tuning approaches rely on instrumentation and (predominantly) post-mortem analysis of the resulting performance data to identify and correct performance bottlenecks.

At exascale, we can expect failures to be more common, suggesting that we must bridge the

chasm separating performance and reliability approaches, addressing performability as a single, runtime, rather than post-mortem concept. This implies the need for real-time measurement of reliability indicators (*i.e.*, transient memory and data transmission errors, behavioral differentials across cores) and performance metrics (computation, networking and storage). By monitoring the temperature of a node, a key insight into the health of the node can be tracked. For example, monitoring the number of retries in the transfer of a message can give an indication of the health of the interconnect. This analysis could be combined with dynamic adaptation, including adaptively selecting checkpointing frequencies, multi-version code execution for verification, automated task retry and autotuning that adapts computation to changing resources. Equally importantly, this will require greater information transparency across software and hardware levels. Coupled with programming model support to describe responses to failure and information from previous executions, compilers should generate multiple code versions that reflect expected failure modes (*e.g.*, restartable models, configurable checkpoints, etc).

7.4.3 Scenario 3: Power Management

Exascale raises power management issues at two levels, both for individual nodes and at the system level. At the node level, this includes managing processor power states, enabling and disabling cores, memory and storage power management, and network interface states. To date, power management has focused largely on processors and voltage/frequency adjustment. However, rather than managing individual cores, managing collections of cores and their chip-stacked memory will become increasingly important, and complicated, as memory will be a major fraction of node power consumption. At the system level, it includes partition management and scheduling and interaction with cooling infrastructure, something not normally considered in high-performance computing software.

Given the wide range of power management scales, it is critical that clear interfaces be defined for information sharing and coordination across hardware, system software, runtime systems and applications. No single level of the hardware/software stack contains all the data needed for power management and optimization. Perhaps equally importantly, these power management issues are deeply intertwined with performance optimization and reliability management (performability), particularly when systems include heterogeneous cores.

To enable intelligent, adaptive decision making, exascale tool systems must include real-time measurement of performance metrics, reliability indicators and power consumption. In turn, decision runtime procedures must be designed to balance the oft-conflicting goals of high performance, bounded power consumption and reliable execution. Similarly, software tools should present not only performance data but power consumption profiles as well. Optimizing for performance alone can push systems beyond their thermal limits and increase the probability of component failure due to thermal stress. Conversely, optimizing for power consumption alone can unduly constrain application performance.

7.4.4 Scenario 4: Debugging

When a user encounters a problem when running an application across a large number of processors, they need to investigate the cause of the problem, where does it manifest itself? Is it a logic error or a system error? If it is a logic error, is it in the serial code, parallel code, or communication code? The application programmer has several options available to them:

1. Run the application again and see if the problem is repeatable

2. Run the application on fewer processors and see if the problem exists
3. Try to schedule interactive time on the same large number of processors to use a debugger to isolate the problem.

Since an exascale system will have billions of threads of execution, no one expects today's debuggers to scale to that level. Therefore, how can the aforementioned enabling technologies be employed to assist the programmer in identifying the problem? One of the most difficult problems is the amount of time a code compiled to be debugged takes to run. The use of the companion process has the potential to significantly reduce that time. With new research, we anticipate that the companion process should be able to use compiler and runtime information to perform the necessary mapping between optimized and unoptimized execution states without slowing down the application threads.

If the program runs the application again and it runs correctly, that does not assure that the application/software is correct. We could actually have the worst kind of error that is typical of a race condition where we have a successful execution one time and an unsuccessful application the next time. A companion process could assist the application developer in identifying the problem in several ways. For example, the companion process could make memory watch points significantly more efficient. By monitoring the executing program's memory, the companion could identify the initial point when a race condition occurs. Then it could halt the application and allow the user to step back to identify which operation actually caused the error.

Another use of a companion process would be to execute an earlier correct version of the program in parallel with the current version to be able to identify the point in time when the two versions differ. This type of analysis is particularly valuable when the programmer is restructuring a currently running application to perform more effectively. Development of a new age of debuggers which have access to a companion process would open up a wide range of additional capabilities to quickly identify error, even when running large scale applications.

A companion process could improve the performance of differential debuggers. With such a companion process, the application could be dynamically checkpointed when the user identified an abnormality. The user could then examine the contents of the application's memory and quickly determine the cause of the problem. Plotting the contents of an array in real time would allow the user to visually identify where abnormal computation is taking place, perhaps at a boundary that is not properly being handled in the application.

7.5 Summary

This chapter has highlighted three key areas of innovation in the software stack needed to develop useable tools that truly enhance programmer productivity in the complexities of an exascale regime. These three areas of innovation are: (1) scalable data collection and analysis; (2) companion computations; and, (3) auto-tuning. Using four scenarios highlighting how tools could enhance programmer productivity, we examine how these areas would facilitate interaction with the application developer and the rest of the software stack. While a lengthy discussion of tool research is beyond the scope of this report, we describe in more detail the set of tools currently used for petascale or envisioned for exascale in Appendix A.2.

Chapter 8

Technical Approach

In this chapter, we outline key elements of a technical approach for Extreme Scale software. The aim of this chapter is to provide examples that are indicative of the kind of software technologies that will be needed to address the Concurrency and Energy Efficiency challenges of Extreme Scale systems, without prescribing specific solutions. It is expected that the community will create shared open source components through efforts such as the International Exascale Software Project [56] that can be leveraged in building Extreme Scale software stacks. Section 8.1 highlights the importance of *software-hardware interfaces* in an Extreme Scale system. Section 8.2 identifies opportunities for addressing Concurrency and Energy Efficiency challenges through *software-hardware co-design*. Section 8.3 discusses the importance of *deconstructed operating systems* in the future OS roadmap for Extreme Scale systems. Section 8.4 presents a vision for Extreme Scale system software based on the notions of a *global OS* and *self-aware computing* [36]. Finally, Section 8.5 describes an example execution model and technical approach, in an effort to encourage the community to think of breakthrough approaches for building Extreme Scale software.

8.1 Software-Hardware Interfaces in an Extreme Scale System

Figure 8.1 shows a notional structure for software and hardware interfaces in an Extreme Scale system. The main motivation for these interfaces is that we expect optimization of Concurrency and Energy Efficiency to be best achieved by graceful cooperation among software and hardware layers. The separation between software and hardware layers is specified as a “Hardware API” — we refer to this interface as an “API” to emphasize the fact that hardware interfaces should not look different from software interfaces from the application viewpoint. As shown in Figure 8.1, there can be multiple levels of information flow within and across software and hardware layers. Some examples of this information flow are as follows:

Programming language supplying data access pattern information to compiler: Programming languages do not often provide the means to express many facts that the programmer may know about their application, such as data access patterns. For example, a programmer may write a library routine for generality such that it can handle both sparse and dense array arguments. However, if the routine is often used for dense linear algebra, a hint like `#pragma expect stride 1` could help the compiler generate more efficient code in the common case. These sorts of pragmas were commonly supported in languages and compilers for old vector machines, but are rarely found on modern cluster platforms. If the programmer does not know what data access patterns the code engenders (perhaps the programmer responsible for code tuning/maintenance did not write the code originally), they should be able to use a tool like

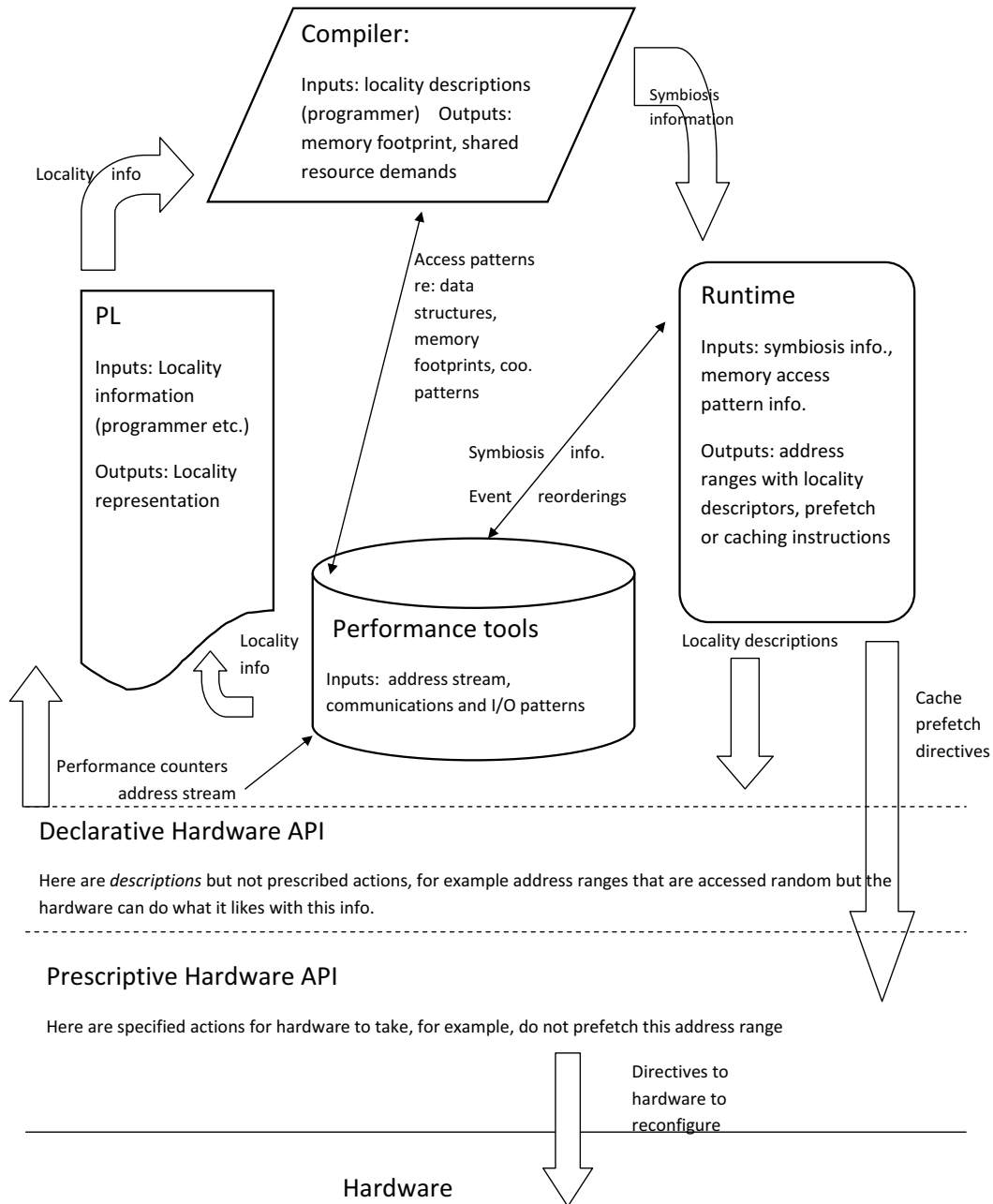


Figure 8.1: Software and Hardware Interfaces in an Extreme Scale System (Notional)

September 14, 2009

Page 79

ECSS Report

a memory access profiler to discover the pattern, and then insert a pragma to capture what has been learned. This scenario involves a feed-back loop among the programmer, compiler, and memory tracer. It should be noted that trace-based recompilation is available today but memory access pattern information for the most part is not available today (a missing, but somewhat trivial to supply hardware API feature).

Compiler providing scheduling information to runtime system: Compilers invest a lot of effort to discover the control flow, instruction mixes, and in many cases the data access patterns and memory footprints of an application, then throw this information away after code generation. In the meantime if the runtime knew the computational demands of the programs being handed to it, it could reason about resource conflicts, and (say) explore symbiotic scheduling that avoids conflicts on shared resources. For example, by binding a non-memory-intensive and a memory-intensive thread together, rather than two memory intensive threads together on the same shared memory node, cache thrashing and memory bandwidth competition could be reduced.

Runtime system controlling data-motion hardware at a fine level of granularity: Armed with information from the user and compiler about what data access patterns to expect, the runtime should be able to ask the hardware to cache certain data structures, not cache certain others, and also should be able to control prefetching and other data motion operations. Based on this information conveyed in a hardware API, a smart memory controller should then be able to avoid moving long cache lines, big memory pages, and substantial disk blocks when only one element of them is likely to be needed.

Performance tools: Memory address stream information (including communications and disk I/O) can be used by performance tools to carry out very simple data analyses (*e.g.*, identifying data structure access patterns) or highly sophisticated analyses (*e.g.*, identifying possible task reorderings that preserve semantics while improving locality) and reflect this back to the appropriate level. In Figure 8.1, the performance tools sit in the middle, observe, and inform, all the other levels.

Declarative Hardware API: This high-level API describes properties of data structures or computations that hardware may choose to take advantage of, for example that a data structure is accessed randomly, or that a computation is memory intensive. The API is also used to support queries on the choices made by the hardware that go far beyond today's hardware performance monitor interfaces *e.g.*, querying the cache management policies used by the hardware for a given address range, based on the declarative information received by the software.

Further examples of the high-level declarative API include:

- Performance profiling requests that include identification of events to be counted and sampled, and interface for software to collect information on performance events such as AMD's proposed interface for Lightweight Profiling.
- Resilience information that includes identification of threads with lower resilience requirements *e.g.*, for which software can perform error detection and recovery so hardware does not have to.

Prescriptive Hardware API The declarative hardware API establishes information flow between software and hardware, does not enable the software to directly control the hardware's

actions. In contrast, the lower level prescriptive hardware API, describes actions that the hardware should or should not take. For example, the hardware can be told to not cache data in a certain address range, or that it should voltage-scale the processor when running a particular region of code. The lower level API should allow hardware to reflect up hardware performance counter information including access to physical or virtual address (something not the case today) without high overhead; this feature alone would greatly expand the capability of performance tools to carry out analysis that in turn can feedback to all layers of software, and back to hardware via API, to improve parallel and energy efficiency. Further elements of the low level API could include:

- Memory hierarchy configuration parameters *e.g.*,
 - Cache sizes, line lengths, degree of associativity
 - Register file sizes, data widths
 - Memory access patterns
 - Address ranges that should bypass cache
 - Address ranges that require hardware coherence
 - Address ranges for which coherence will be managed by software
 - Address ranges with values that are guaranteed to be read-only (immutable) for certain application phases
 - Network bandwidth partitioning for different forms of data movement and communication
 - PGAS
 - RDMA
 - Message passing
 - Stream processing
 - Other network reconfigurability parameters including topology and packet size
- Power management
- Frequency scaling (smaller time constant, but constrained by voltage)
 - Voltage scaling (larger time constant)
 - Issue width management
 - speculation control
 - core power cycling temperature-based power policies

As different software stack components are developed for Extreme Scale systems, it will be desirable to reuse components across the three classes of systems. To that end, Table 8.1 summarizes the key similarities and differences among software stack components across all three extreme scale system classes.

8.2 Opportunities for Software-Hardware Co-Design

We believe that software-hardware co-design will be a critical necessity for Extreme Scale systems, in addition to the interfaces outlined in the previous section. This form of co-design has been essential for *vector parallelism* [85] in current and past systems, and is also being explored for scalable approaches to mutual exclusion using *transactional memory* [92]. In this section, we discuss a few additional examples of software runtime capabilities that will be necessary for future Extreme Scale systems, and examine how they can be made more effective with software-hardware co-design.

September 14, 2009

Page 81

ECSS Report

Exascale Software Study

Software Stack Components	Data Center	Departmental	Embedded
Operating Systems	Capability: restricted high-end applications	Feature-rich: broader mix of applications, ISV software	Deterministic, Real time
File Systems	Scalability: Large number of files	Enterprise file system	Flat, simple, limited
I/O	Some apps are massively I/O bound	Standard network I/O, richer set of I/O drivers	I/O bound
Resilience Runtime	Intelligent checkpointing	Could approach Enterprise mission-critical	Fault tolerant apps, Replication for mission- critical
Task Scheduling Runtime	Featherweight asynchronous tasks	Lightweight asynchronous tasks	Deterministic, Real time
Memory Management Runtime	Typically, single job at a time, limited # large address spaces	More sharing, larger # address spaces	Single job
Intra-System Communication Runtime	Massive # nodes, special communication hardware, Special controls for data movement, Latency on critical path	Commodity	Small number of nodes, application-specific data movement
Power Management Runtime	Adaptive, large-scale power management	OS-driven	Application-driven (platform-limited)
Profiling & Monitoring Runtime	Scalable, Online analysis/aggregation, anomaly detection	Should be accessible to non-experts	Lightweight, predictable, fixed, time intervals
On-chip communication Runtime	Locality-aware dynamic data movement and load balancing	Locality-aware dynamic data movement and load balancing	Support for dynamic, systolic, and statically scheduled communications
Static & Dynamic Compilers	Support for new task & locality constructs, use of spare cores, dynamic optimization	Support for legacy codes and their migration	System-level optimization, more heterogeneity
Programming models	Expression of new task & locality constructs, ultra-fine-grain parallelism	Expression of new task & locality constructs, very fine-grain parallelism, Should be accessible to non-experts	Expression of new task & locality constructs, real-time constraints

Table 8.1: Key Software Stack Component Similarities and Differences for Extreme Scale System Classes

8.2.1 Scheduling dynamic parallelism with fine-grained tasks

As discussed in Chapter 5, it is important to ensure that the intrinsic parallelism in a program can be expressed at the finest level possible *e.g.*, at the statement or expression level, and that the compiler and runtime system can then exploit the subset of parallelism that is useful for a given target machine. There have been multiple proposals for expressing fine-grained parallelism *e.g.*, statement-level *spawn* [43] or *async* [50] operations, expression-level *future* [76] operations, and operator-level data flow graphs [54, 128]. These operations for fine-grained parallelism are in stark contrast with the *bulk-synchronous parallel model* [139]. While profile-directed compile-time partitioning can be used to optimize the granularity of fine-grained tasks in certain cases [116, 117], in general the runtime system also needs to participate in the partitioning so as to best adapt to unpredictable execution times. A classic approach to runtime partitioning is *lazy task creation* [104], which has been extended into *work-stealing runtimes* for fine-grained tasks [65, 73]. A work-stealing runtime system creates a fixed number of worker threads, with one local double-ended queue (deque) per worker. Each worker repeatedly picks up work from a deque of lightweight tasks using scheduling policies that are designed to achieve good load balance while bounding the size of the deque. This approach has been shown to yield scalability that is orders-of-magnitude superior to the scalability achieved if each task were to be created as a thread at the OS level.

However, there are still significant overheads that remain in a software-only approach, that will likely prevent it from being usable at Extreme Scale. These overheads involve locking operations, and in the case of nonblocking algorithms involve spin loops on shared memory locations with their accompanying cache consistency overheads. As mentioned earlier, these overheads are especially important because they occur on critical paths in parallel programs. *Hardware support* for shared queue data structures can result in orders-of-magnitude reductions in scheduling overheads and scalability bottlenecks, while still retaining the flexibility of task scheduling policies in software. As mentioned in Section 8.2.4, hardware support for shared queues can have other uses as well in Extreme Scale systems.

Another source of overhead in task scheduling lies in the operations that need to be performed on the fast path to save local variables, so as to ensure that the task can be resumed on a separate worker from the one that it started on (if needed). A software-only approach introduces word-at-a-time store instructions to save the local variables, and some of these stores are often redundant. In contrast, hardware support for saving and restoring local variables (as in calling conventions) can help reduce this overhead that occurs on the fast path.

8.2.2 Distribution and co-location of tasks and data

Another candidate for software-hardware co-design pertains to distribution and co-location of tasks and data, which is one mechanism that can be used in support of locality optimization. As observed throughout this report, it will be critical to optimize vertical locality so as to satisfy the energy constraints of Extreme Scale systems. Runtime systems for programming languages such as UPC [60] and Co-Array Fortran [109] that are based on a *Partitioned Global Address Space* (PGAS) model include the notion of virtual *home location* for each shared datum. The more recent HPCS languages extend this notion of home locations to computational tasks, as in Chapel's *locales* [53] and X10's *places* [50], so as to enable tasks to be shipped to data, data to be shipped to tasks, or any meet-in-the-middle combination thereof. The translation from global to local addresses is a major source of overhead in a software-only approach to implementing such languages, along with the communications that accompany non-local accesses. Thus, it becomes important for a compiler for such languages to perform redundancy elimination on address computations, to coalesce

contiguous accesses into a single communication operation, and to overlap communication with computation [148]. However, many of these optimizations can still remain in a software-only approach after compiler optimization. Opportunities for software-hardware co-design include the use of translation buffers to accelerate virtual-to-physical address translations, and DMA-like hardware support to reduce the processor overhead of data transfers.

8.2.3 Collective and point-to-point synchronization with dynamic parallelism

As discussed in Section 5.4, the fine-grained parallelism intrinsic to a program may need to be accompanied by fine-grained collective and point-to-point synchronization among dynamically varying sets of fine-grained tasks. These synchronization structures may be irregular, and tasks are permitted to dynamically join or leave these structures as in the *phasers* construct [125]. Further, it is usually desirable to augment the synchronization structures with communication for reductions [124], collectives, and systolic computations. As discussed in Section 8.2.1, synchronization structures always represent good candidates for software-hardware co-design since a software-only approaches for synchronization suffer from unnecessary cache consistency and serialization bottlenecks. Hardware support (*e.g.*, in the form of counting semaphores) can be used to reduce the overhead of inter-core synchronization, and extensions in the form of register-level inter-core communication (*e.g.*, as in the Raw project [94]) can reduce the overhead of communication. Further, the use of a single master task to perform a reduction in software can be a scalability bottleneck, and a software-only approach to creating combining trees incurs high setup and tear-down overhead. Instead, hardware support for combining synchronization and reductions will greatly reduce the overhead of collective and point-to-point synchronization with dynamic parallelism.

8.2.4 Producer-consumer parallelism

Another common idiom in fine-grained parallel programs is that of producer-consumer parallelism. In this model, a single-writer task serves as the producer of a datum for multiple readers. To accomplish this, the writer task typically stores its result in a designated location, and the reader tasks block when they request the result (if the result is not ready). In the case of *futures* [76], the execution of the writer task may (optionally) be deferred till the datum's value is requested by one of the readers. Once again, we observe that a software-only approach suffers cache consistency and serialization bottlenecks, and hardware support can be used to reduce these bottlenecks. A classical example of hardware support in this area is the *full-empty bit*, but there may be many other variations. Also, in many cases, the location on which the producers and consumers wish to synchronize may be designated by a tag rather than an address. Hardware support to accelerate the translation of tags to addresses can be very useful. Intel's Concurrent Collections (CnC) [46, 47] is an example of a high-level programming model that relies heavily on producer-consumer parallelism and that would benefit greatly from any hardware support.

8.3 Deconstructed Operating Systems

Operating systems must be refactored (deconstructed) to offer more flexible resource management and runtime support for parallel execution models with the focus on exposing system resource usage policies to the various level of the programming stack. The overall goal of a deconstructed OS would be to allow the application to compose the best resource usage policies for its particular needs and to adapt to system scale and load. Policy control should be hierarchical, with different levels of abstraction depending on their consumer. For example, a future communication scheduling

mechanism could expose to the libraries/compiler explicit control over message sizes and ordering, while exposing to the application level/programmer only abstract policies like “long routes first”. Adaptation can take form of Quality of Service mechanisms or migration for communication locality.

Previous experience with compiler and runtime optimizations for PGAS languages indicates that lightweight control over OS mechanisms is not sufficient for good performance and additional control is required over the policies that guide the management of these mechanisms. Looking beyond PGAS languages, even more control of resource management will be necessary to support the kind of novel execution strategies required for Exascale applications. Current OS designs favor generic policies, *e.g.*, preemptive thread scheduling or Least Recently Used page replacement, which have been selected as being the least common denominator for commercial workloads. In contrast, the execution models required for structured parallel algorithms in scientific computing applications are less diverse, usually require cooperation among computing entities and exhibit a relatively ordered execution imposed by data/task dependence.

8.3.1 Role of Hypervisors in a Deconstructed OS

The resources (such as memory, bandwidth, and power) in an extreme scale system are expected to be highly constrained. Exascale hardware technology is also envisioned to be highly memory constrained. Therefore, rather than a full OS model, there may be substantial benefit from an “exokernel” model [61] where applications are bundled with only the necessary OS functions linked in to the application that run in their own virtual machines (VM) container, relying on the hypervisor or VM container for protection, resource sharing, and management of Quality of Service (QoS) guarantees. We will refer to these protection mechanisms as an “application container” because there are a number of technologies including hypervisors, VMMs, and runtime environments such as Singularity that can implement this kind of isolation in a spatially partitioned CMP. The primary roles of the application container are to manage partitioning of hardware resources, including physical processors, physical memory, and memory and interconnect bandwidths, while the runtime layer above the application container will have complete control over scheduling and virtualization, if any. For example, in an SPMD execution layer used in UPC, there is no need for processor virtualization, while for dynamic threading used in the DARPA HPCS languages, a lightweight user-space thread scheduler that can be directly controlled by the application or runtime would be beneficial.

One approach to both operating systems and runtimes for parallel execution is to deconstruct conventional functionality into primitive mechanisms that software can compose to meet application needs. A traditional OS is designed to multiplex a large number of sequential jobs onto a small number of processors, with virtual machine monitors (VMMs) adding another layer of virtualization. An alternative approach is to explore the usage of a very thin hypervisor layer that exports spatial hardware partitions to application-level software. These virtual machines allow each parallel application to use custom processor schedulers without fighting fixed policies in OS/VMM stacks. The hypervisor supports hierarchical partitioning, with mechanisms to allow parent partitions to swap child partitions to memory, and partitions can communicate either through protected shared memory or messaging. Traditional OS functions are provided as unprivileged libraries or in separate partitions.

For example, device drivers run in separate partitions for robustness, and to isolate parallel program performance from I/O service events. An alternative “deconstructed” architecture would enable partitioning not only of cores and on-chip/off-chip memory but also of the communication bandwidth among these components, with QoS guarantees for each partition. The resulting performance predictability improves parallel program performance, simplifies code (auto)tuning and

dynamic load balancing, and supports real-time applications.

Hypervisors, VMM-based application containers, and various code-rewriting systems offer a thin protection layer that can be used to support desired capabilities for massively parallel CMP-based systems summarized in the following subsections.

8.3.1.1 Minimalism, Modularity, Mediation

A thin protection layer is needed on a CMP to prevent hardware state corruption, but otherwise offer bare-metal access to the underlying hardware wherever possible. Many system facilities will be linked at user level as a set of optional systems libraries. Hardware protection mechanisms will allow direct, user-level access to facilities such as networking and I/O through the lightweight protected messaging layer. Parallel applications will be given bare metal partitions of processors that are dedicated to the given application for sufficiently long periods to provide performance predictability. The primary roles of the protection layer are to manage partitioning of hardware resources, including physical processors, physical memory, and memory and interconnect bandwidths, while the runtime layer above the protection layer will have complete control over scheduling and virtualization, if any. There are many lessons to be learned from K42 [39], the MIT Exokernel [61], and embedded operating systems such as VxWorks [34] regarding approaches to efficient and modular system services.

The thin protection layer (possibly a hypervisor or VMM-based application container) will play a role in mediating concurrent access to devices to ensure fair sharing of resources. Although software functions can be virtualized through replication, hardware devices are finite and access to them by multiple hardware components must be managed. This will include Quality Of Service guarantees for access to certain rationed resources such as memory or network bandwidth.

8.3.1.2 Isolation

Groups of processors can be combined into protected partitions. Boundaries will be enforced through hardware mechanisms restricting, for example, sharing of memory across partitions. Messaging between partitions can be restricted based on a flexible, tag-based access control mechanism. OS functionality such as device drivers and file systems will be spatially distributed rather than time-multiplexed; we refer to this as spatial partitioning. This approach works in synergy with the sidecore techniques [91], which allow an application to vector OS or driver functions to execute on free cores rather than forcing a context-switch on the core running the application.

8.3.1.3 Safe User-Level Messaging

Messages will be used to cross protection domains rather than more traditional trap-based mechanisms. Through hardware mechanism and/or static analysis, applications will have direct, user-level access to network interfaces and DMA. Further, fast exception handling and hardware dispatch of active message handlers will permit low overhead communication without polling. Most traditional system-calls will translate into messages to remote cores (other system-calls will be to linked system libraries) [91].

8.3.2 Related Work

Operating system research is an infrastructure-intensive process with a long initial development time. Traditional operating systems have a monolithic design with little or no control over the internals exposed to the application or user level. Novel programming models need to demonstrate

clear performance and productivity advantages over the established paradigms in order to have a chance for widespread adoption. Their efficiency can be greatly improved when having access to fine application level control over functionality provided by the system software stack (OS). The same fine-level of control is beneficial to established execution models of existing parallel runtime environments.

There are several DOE sponsored ongoing research projects related to operating systems design. The ZeptoOS [35] project distinguishes between service node and compute node kernels and provides tool for OS instrumentation and understanding of the interaction between the OS and the application layer. The Right-Weight Kernels [103] project focuses on a judicious selection of OS level services in order to diminish OS interference. The K42 research project focuses in parallelizing the OS itself and providing abstractions for overall system adaptation at scale. A common characteristic of these projects is the focus on improving overall system behavior and reducing OS interference [113] and putting more resources under the application and user space control while maintaining fault isolation and security. They mostly explore the opportunities offered by lightweight kernels and focus on kernel level mechanisms for resource control: CPU, memory, and network.

8.4 Global OS and Self-Aware Computing

8.4.1 Services for Adaptation at Large System Scale

Optimization and execution decisions should be made based both on the instantaneous system state and global knowledge about the application behavior. For example, data transfer mechanisms should be instantiated based on current load and the logical communication topology of the application. This unfolds into the following research directions:

- Develop automated frameworks for exploration of system characteristics to understand scaling behavior. Scaling behavior is determined by a combination of the hardware characteristics of the system and the way the application uses the hardware. The result is a highly dimensional parameter space that is not fully explored by current benchmarking techniques. Furthermore, due to restricted access to large-scale systems, the benchmark process today provides in some instances statistically unsound data.
- Develop policies and mechanisms for adaptation. These will include:
 1. Automatic mechanisms to enforce flow control and avoid congestion intra and inter node, exposing and using QoS primitives in the application
 2. Intra-node mechanisms for communication scheduling such as scheduling of communication operations based on layout, *e.g.*, long-range communication has priority over short range communication or coalescing of communication operations that target the same node.
 3. Selection of the best communication primitives based on load and application requirements, *e.g.*, selection between active message or pipelining based implementations for applications that process a large number of disjoint remote memory regions.
- Develop mechanisms to expose application behavior to the adaptation mechanisms. This may include exploration of two different alternatives:
 1. offline instrumentation and feedback

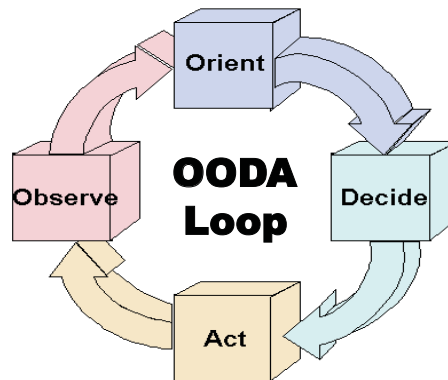


Figure 8.2: OODA Loop

2. continuous monitoring and feedback loop.

- Explore formal methods to describe the communication requirements and behavior of an application and annotated execution models.

8.4.2 Sensors and Actuators

Sensors will include *machine profile sensors* that can determine properties of the target machine on which the application runs including capacities and bandwidths of the memory hierarchy, to allow the application to adapt to the hardware configuration on which it runs. Further sensors may determine dynamically the level of contention from the hardware/runtime and allow an application to deploy an algorithm that is more efficient when bandwidths to shared resources drop.

Actuators from the application side will include *application signatures* to represent the resource requirements of the application to the other levels of the stack and the hardware. For example an application's memory footprint will be an actuator that can cause the runtime to allocate sufficient memory; memory access patterns may cause the runtime or compiler to optimize prefetching policies for the specific application. Information about symbiosis (the impact of the application upon shared resources) may be used by the O/S to choose co-scheduling partners for the application.

8.4.3 Self-Aware Systems

Current operating systems have pre-programmed behaviors that are based on guesses about resource availability. As a result, they are ill-suited to complex multicore systems and result in sub-optimal performance in the face of changing conditions. In contrast, it will be desirable for the OS to behave like a *self-aware system* that “learns” to address a particular problem by building self-performance models, responding to user goals, and adapting to changing goals, resources, models, operating conditions, and even to failures.

Figure 8.2 illustrates the basic Observe-Orient-Decide-Act (OODA) loop of a self-aware system. Thus, a self-aware system is

- Introspective — it observes itself, reflects on its behavior, and learns.

September 14, 2009

Page 88

ECSS Report

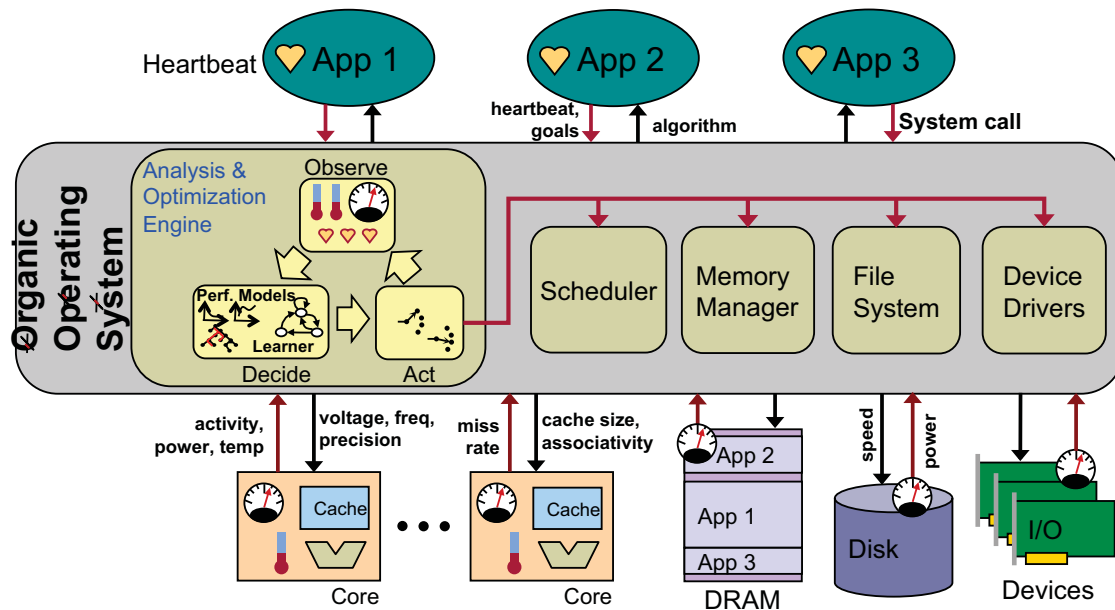


Figure 8.3: Organic Operating System (Notional)

- Goal-oriented — ideally, the system's client only specifies the goal, and it is the system's job to figure out how to get there
- Adaptive — the system analyzes the observations, computing the delta between the goal and observed state, and takes actions to optimize its behavior
- Self-healing — the system continues to function through faults and degrades gracefully
- Approximate — the system does not expend any more effort than necessary to meet goals

As an example of a self-aware OS, consider the notional Organic Operating System (OOS) shown in Figure 8.3. The Observe and Orient steps are accomplished via a number of new *sensor* interfaces that will need to be added to all software and hardware components of an Extreme Scale system (yet another example of software-hardware co-design). These observations include processor characteristics such as performance, energy, miss rates, queue lengths, resource utilizations as well as physical characteristics such as temperature. The Decide and Act steps are enabled by a new *actuator* interfaces that enable the system to control application behavior at a number of levels such as number of allocated, cache configurations, scheduler policies, clock frequencies, and numerical precision desired. Thus, a separation of concerns is achieved between the application and the OOS where the application communicates goals and options to the OOS, and the OOS uses component performance models to decide how best to meet goals under given system constraints.

8.5 Silver: An Example Execution Model and Technical Approach for Extreme Scale Systems

This section offers an example of one possible Exascale software stack, referred to as “Silver”, to serve as an exemplar for the general class of transformative software strategies to address the broad challenges of extreme scale computing. The design of Silver is based on the premise that any extreme scale software stack must address four critical obstacles to scalability: starvation or insufficient program parallelism, overhead or critical path management work, latency to main memory and across system, and contention for concurrent service requests to shared resources. Each of these critical factors is influenced and impacted by design choices made at every layer of the computing stack. Furthermore, the design of any stack layer interrelates with all of the others, but in particular with adjacent layers. Additional performance and quality of service factors to be addressed include reliability, availability, programmability, and cost. The future generation extreme scale system software stack may take on any one of many forms and understanding and evolving these will require a vision that recognizes these needs.

8.5.1 Silver Execution Model

The unifying set of principles for Silver is a model of computation that is a synthesis of semantic constructs, policies, and mechanisms comprising the logical organization and operation of a parallel computation to be performed. The Silver model strives to

1. provide an abstraction of parallel computation that exposes and exploits a high degree of algorithm concurrency, particularly that available from dynamic directed graph structure-based applications,
2. enable intrinsic latency hiding through automatic overlap of computation and communication through message-driven work-queue multithreaded execution,
3. minimize impact of synchronization and other overheads for efficient scalable execution through lightweight object-oriented semantics,
4. support dynamic global address space scheduling for adaptive resource management, and
5. unify heterogeneous structure computing for diversity of processing modalities and exploitation of accelerator micro-architectures.

The Silver model supports a work-queue model. Threads are created locally by other threads or remotely by message-driven mechanisms (parcels) permitting work to be moved to the data. Threads are organized within the contexts of parallel processes, each spanning potentially many local domains (localities). Synchronization among threads is achieved through local control objects (LCOs), providing a plethora of mechanisms from simple mutex to more complex dataflow and future constructs. The message-driven model serves logical destinations or physical destinations, accelerators, or output devices. Percolation supports pre-staging of data and executables at remote resources to hide latency to these separate computing elements by overlapping communications with computing and to avoid their overhead costs, needed to take advantage of heterogeneous processors and precious resources.

8.5.2 Silver Stack

Silver addresses the challenges of the design and operation of a system stack for a new generation of ultra scalable computer systems capable of extreme scale performance with technologies nearing nanoscale feature size and the end of Moore's Law. The overall system stack as structured by Silver is deceptively similar to more conventional systems and comprises seven layers:

1. Driver applications — a focus on the emerging class of applications incorporating very large sparse, irregular, and time varying data structures including science and informatics will drive co-design across the system stack with a set of selected problems employed for experimental pursuits.
2. Programming models, methods, and tools — provides at least three distinct but interrelated user-level programming interfaces that serve as protocols for program requirements to system resources and control mechanisms. These variations include a library of service calls, mapping to this of conventional (legacy models), and an advanced low-level language optimized around the needs of the execution model and the directed graph based applications.
3. Compiler strategies and design — combines compile time analysis, adaptive learning, just-in-time modules, interface to advanced runtime mechanisms to control application parallelism, data management and distribution, and physical resource allocation.
4. Runtime system software — provides dynamic application execution scheduling, synchronization, and name space management under the control of the compiler and utilizing OS supplied resources.
5. Operating system - manages the hardware resources, by providing control of user processes, virtual memory allocation and name spaces, provides essential services to programs and runtime system, and recovery response in the presence of hardware and software faults.
6. Architecture - comprises both system level and micro-architecture utilizing the underlying enabling technologies to provide most effective and scalable computing for the essential modalities exhibited by the target applications by means of the software and programming layers.
7. Enabling technologies — defines a technology roadmap that will establish the bounding conditions, opportunities, and requirements that have to be satisfied to realize efficient, scalable computation.

For this Exascale Software Study the first and last layers of the System Stack provide boundary conditions and the penultimate layer, architecture, is a flexible supporting medium of computation that will be influenced by as well as influences the other layers of the software stack. We elaborate on items 2-6 below.

8.5.2.1 Programming Methods

An important part of the programming layer of the stack is to ensure that existing applications (so-called legacy applications) which embody a great deal of knowledge and development can be adapted to the Silver platform. This means adapting codes that use the Message Passing Interface (MPI). There are several approaches to this. First, it needs to be demonstrated that MPI programs can run well on this system. The MPI Forum has begun developing the next generation of the MPI specification. This layer of the stack must reflect the extensions to MPI that will define MPI-3.

Silver-Threads will be a language for the Silver programming model. It is expected that this effort will result in a new language, not bound by the semantic decisions of any existing languages. This programming interface will be largely defined by what it chooses to expose to, and what it chooses to hide from, the programmer. Low level constructs, such as synchronization through local control objects and messaging via parcels, will be hidden by high-level descriptions of tasks. The programmer should not be concerned with the exact locations of resources: all resources will appear local, with the address translation and parcel system handling migration of flow control between localities. Silver-Threads should take advantage of the advanced namespace management in the Silver execution model, including the DGAS component and the promotion of processes and threads to first class objects. The latter is particularly interesting, as it will allow the program to "reason about its execution." Another facet of Silver-Threads programming language will be the ability to specify optional heuristics, for providing "hints" to the runtime system regarding expectations for load balancing and resource management.

8.5.2.2 Compilation Methods and Tools

Substantial amount of research in parallelization has focused on "regular" programs which manipulate dense matrices. Unfortunately, exploiting parallelism in "irregular" programs — such as those that operate on lists, trees and graphs — is much harder. The amount of parallelism in dynamic graph-based computations depends on the input data and is not known until execution. The Silver compiler approach to this problem is to develop a suite of dynamic locality-aware partitioning techniques that allow hiding of global system latency. These techniques exploit significant computation-communication overlap. For dynamic graph-based computations, effective dynamic mapping and remapping of data will be crucial to realizing high-levels of performance. The Silver approach is to first develop mapping and remapping techniques, which will be augmented with strategies for deciding when it is effective to remap data. These solutions will be integrated seamlessly with the message-driven threaded-execution model that is at the heart of Silver Execution Model. Another task for the compiler is to extract high levels of performance from the underlying heterogeneous architecture containing such diverse themes as streaming and PIM micro-architectures. We will develop program partitioning strategies aimed at this.

8.5.2.3 Runtime Systems

The Silver runtime system is designed to be a modular, feature-complete, and performance-oriented representation of the Silver execution model on conventional (Linux based) architectures, offering an alternative to conventional computation models, such as MPI. It will also be targeted to the advanced Silver OS described below. The Silver model is intrinsically latency hiding, delivering an abundance of parallelism in within a hierarchical distributed global shared name-space environment. This allows the Silver runtime to provide a multi-threaded, message-driven, split-phase transaction, non-cache coherent distributed shared memory programming model using futures based synchronization. It is a second-generation runtime system for possibly heterogeneous platforms designed for applications handling very large dynamic distributed graphs. It can be implemented using C++ which allows combining well designed modularity with efficient runtime performance. This enables

- software optimizations at component level,
- policy based parameterization and configuration at compile time and runtime, and
- high portability to new hardware and systems architectures. The Silver runtime will support advanced dynamic application frameworks such as Charm++.

8.5.2.4 Operating System

The Silver operating system strategy is a two-prong approach with 1) a limited-scale extended conventional path based on emerging community-wide multicore Linux solutions, and 2) a second transformative approach to ultra scalable operating system design based on synergistic integration of lightweight kernel modules; the latter called “Silver-OS” or “SOS”. The first permits early integration of the Silver runtime system on conventional platforms to support the directed graph oriented advanced programming models and applications for near term use and experimentation. The second will empower ultra scalability to hundreds of millions of cores of diverse structure and operational modalities for extreme scale sustained performance implemented with evolving semi-conductor and optical technologies culminating with nano-scale devices. Integration of incremental enhancements of Linux will be achieved by the runtime and compiler layers.

Silver-OS is a new strategy to provide global unification of system-wide services such as memory management, thread management, and global communications for systems comprising billion-way scale parallelism. The innovative concept to be developed and applied is that of synergistic protocols between like services on separate compute nodes. This synergistic control model will enable the synthesis of the physically disparate elements of like functionality to be logically integrated into single global functions that may be dynamically managed and globally optimized in their allocation and operation. The overriding transformative benefit from this unique strategy is its scalability with fixed design complexity. Thus global unification is accomplished and is logically provided to the user through local interfaces among global functional services rather than global interfaces among local functional services. This revolutionizes the system support for user programs and parallel programming language design for scalable systems to potentially hundreds of millions of cores required for Exaflops scale system operation. Distributed global functionality to be realized through synergistic lightweight kernel agents include:

- distributed global address space allocation and address translation,
- parallel process multiple locality assignment and environment context management,
- parallel thread instantiation, suspension, and context switching,
- active message (parcels) creation, routing, buffering, and acquisition,
- I/O including interactive channels and file system interface, and
- process isolation and protection.

The Silver OS will use these lightweight distributed interoperative agents to support a POSIX equivalent API with lower overhead and greater scalability as well as provide the more dynamic functionality required by future generation programming models (*e.g.*, Silver-Threads) and applications.

8.5.2.5 Silver Architecture

Silver is a conceptual system (hardware and software stack) devised to support parallel computation guided by the governing principles of the Silver execution model of computation to effectively exploit future trends of enabling technologies. The Silver architecture that is hypothesized as a starting point for exploration is based on the Silver execution model and is envisioned as a heterogeneous structure of two classes of micro-architectures optimized for two modalities, or operating points, respectively. Like conventional architectures, temporal locality is a dominant dimension.

September 14, 2009

Page 93

ECSS Report

But while conventional architectures assume good temporal locality and commit the majority of its resources to cache hierarchies, the Silver architecture recognizes that computations exhibit disparate operational modalities. Examples of high temporal locality compute elements include the Stanford Merimac, the UT-Austin Trips, and the Cray X1 PVP. A low/no temporal locality class of architecture elements considered as part of the Silver stack architecture layer is an advanced PIM architecture to accelerate data intensive computation, such as dynamic directed graph processing, exhibiting low temporal locality resulting in little data reuse and poor cache behavior. The Silver-PIM is a lightweight multi-threaded core that exploits the low latency and high bandwidth achieved through direct access to the wide row buffer of memory banks. Silver-PIM incorporates distributed global address space (DGAS) virtual to physical address translation, and message-driven thread instantiation with efficient compound atomic operations on structs for efficient local control object synchronization overhead. It supports the message-driven work-queue method for Silver execution Model and messages for system-wide intrinsic latency hiding.

Chapter 9

Conclusions

There are several reasons for paying attention to software in the development of Extreme Scale systems. First, the extreme scale systems that are projected for the 2015 – 2020 timeframe are dramatically different from today’s Petascale systems and will require correspondingly fundamental changes in the execution model and structure of system software (both of which have remained relatively stagnant during the last two decades). Second, while there has been significant innovation at the hardware and system level for today’s Petascale systems, previous approaches have not paid much attention to the co-design of multiple levels in the system software stack (OS, runtime, compiler, libraries, application frameworks) that is needed for extreme scale systems. Third, while certain execution models such as Map-Reduce in cloud computing and CUDA in GPGPU data parallelism have demonstrated large degrees of concurrency, they haven’t demonstrated the ability to deliver 1000× increase in parallelism to a single job with the energy efficiency and strong scaling fraction necessary for Extreme Scale systems.

The starting point for this study was the characterization of Exascale systems in the prior hardware study on “Technology Challenges in Achieving Exascale Systems” [62], summarized in Chapter 2 of this report. In this study, we identified Concurrency, Energy Efficiency and Resiliency as fundamental challenges for Extreme Scale software, and focused on the first two. (The third challenge, Resiliency, is addressed by a companion study.) The Concurrency and Energy challenges are further exacerbated by the lower bytes/ops ratios and the dominant contribution of data movement to energy costs expected in Extreme Scale systems. We observed that the Concurrency and Energy Efficiency challenges have to be addressed at all levels of the software stack, and in conjunction with hardware interfaces and software-hardware co-design.

To better understand the software challenges for Extreme Scale systems, we first introduced a set of desiderata for an Extreme Scale execution model and defined metrics that can be used to compare different software stacks for Extreme Scale systems using *energy-delay* as the foundational metric (Chapter 3). We then studied the challenges and implications in developing applications for extreme scale computing by examining multiple application classes including traditional HPC applications, coupled models, data-intensive, data mining, and real-time applications (Chapter 4). From an application viewpoint, the Concurrency and Energy challenges boil down to the ability to express and manage parallelism and locality in the applications. This chapter concludes that applications can be enabled for exploiting extreme scale hardware by exploring a range of *strong scaling* and *new-era weak scaling* techniques, but only with suitable attention to efficient parallelism and locality.

Given this context, Chapter 5 summarized the challenges in *expressing* parallelism and locality in Extreme Scale software. One of them is the ability to expose all of the intrinsic parallelism and

locality in an application, so as to make the application *forward scalable*. Another is to ensure that this expression of parallelism and locality is *portable* across vertical and horizontal dimension. Additional challenges include *composability* of parallel programs, support for *algorithmic choice* across scale, and support for *heterogeneous hardware*.

The challenges in *managing* parallelism and locality are discussed next in Chapter 6. Since the Operating System provides the foundation for the software stack, a lot of attention was devoted to limitations in current OS structures in addressing Extreme Scale software requirements. OS-related challenges include *parallel scalability*, *spatial partitioning* of OS and application functionality, *direct hardware access* for inter-processor communication, *asynchronous* rather than interrupt-driven events, and *fault isolation*. There are additional challenges in *runtime systems* for scheduling, memory management, communication, performance monitoring, power management, and resiliency, all of which will be built atop future Extreme Scale operating systems. The chapter concludes with challenges in *compilers* and *libraries* for Extreme Scale systems.

Programming tools play an important role in making the development of parallel software more productive. Though tools were not directly in the scope of our study, we recognized from the start of the study that any software stack for Extreme Scale systems must be capable of supporting the tools that we envision will be shipped in that time frame. To that end, Chapter 7 identifies a number of challenges in supporting Extreme Scale tools including *performability*, *scalability*, *abstraction*, *adaptation and autotuning*, *multilevel integration*, and *availability and portability*. This chapter also lists some of the key technologies that will be necessary to address these challenges, as well as six different scenarios that can be used to evaluate the effectiveness of an Extreme Scale system in supporting tools.

Chapter 8 outlines key elements of a technical approach for Extreme Scale software. The aim of this chapter is to provide examples that are indicative of the kind of software technologies that will be needed to address the Concurrency and Energy Efficiency challenges of Extreme Scale systems, without prescribing specific solutions. Section 8.1 highlights the importance of *software-hardware interfaces* in an Extreme Scale system. Section 8.2 goes one step further and identifies opportunities for addressing Concurrency and Energy Efficiency challenges through *software-hardware co-design*. Section 8.3 discusses the importance of *deconstructed operating systems* in the future OS roadmap for Extreme Scale systems. Section 8.4 presents a vision for Extreme Scale system software based on the notions of a *global OS* and *self-aware computing*. Finally, Section 8.5 describes an example execution model and technical approach, in an effort to encourage the community to think of breakthrough approaches for building Extreme Scale software.

Appendix A

Additional Extreme Scale Software Ecosystem Requirements

This chapter summarizes requirements from other components of the software ecosystem that future Extreme Scale software stacks need to be aware of. However, the core technologies needed for these components are considered to be beyond the scope of this study. Many of these topics are being addressed by separate studies and research programs.

A.1 Real-time and Other Specialized Requirements in Embedded Software

Terascale embedded systems based on extreme scale technology will share some of the software stack elements required by departmental and data center systems (Table 8.1), but will also have additional specialized requirements not needed in the other systems. For instance, Figure A.1 (taken from [62]) illustrates the relative importance of technology gaps in different classes of extreme scale systems. This figure suggests that embedded extreme scale systems will need many of the same power and concurrency management features of the software stack as the larger systems, but may not require as many resources devoted to resiliency management. The difference in requirements at the embedded scale was not addressed during this study, however, some of the potential differences are summarized below. Additional studies are under way to identify and characterize requirements particular to the embedded scale in more detail.

The primary additional requirements for embedded systems are *real-time* constraints and restrictions on the deployable form factor. Embedded systems are frequently created for scenarios in which they operate on data that is a direct observation of an external system, obtained via sensors. Often, the external system under observation is an independent physical system that can not be slowed down or accelerated. For a given sensor time resolution, this results in a flow of observed data that is input to the embedded system at a fixed rate. The behavior of the observed system therefore imparts a “real-time” requirement on the embedded computing system. For some applications, it is possible to lower the rate of input to the embedded computing system, resulting in outputs that are of degraded value. Depending on the mission of the embedded computing system, this can impose throughput or latency requirements, or both. The embedded system must perform the desired calculations at a rate that consumes the input data at least as quickly as they are supplied. If the computing is implemented as a pipeline, then each stage of the pipeline must be capable of keeping up with its own data input rate. Some applications (*e.g.*, control, or interactive applications) have an additional latency requirement. For these, throughput high enough to keep

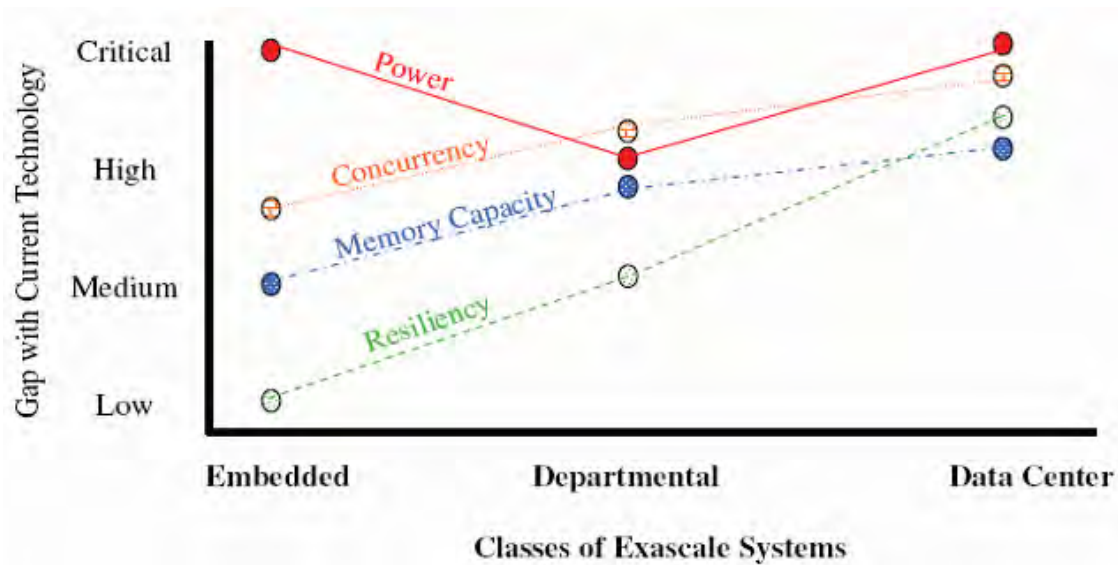


Figure A.1: Technology gaps in Exascale System Classes [62]

pace with the source data is not sufficient — the embedded computing system must also react to its input and produce a stimulus to the external system within some deadline in order to ensure proper behavior of the overall system in which it is embedded.

An important implication of real-time requirements is that, for a given functionality, the utility function for the embedded system versus throughput or latency becomes a step function. This is in contrast to the departmental and data center systems considered in this study. If the throughput is below the minimum required to meet performance goals, or the latency above the maximum, the system fails and has no utility. In many cases, if the throughput is higher than the minimum or the latency is below the maximum, no additional utility is created. In contrast, a non-real-time system still has utility if it falls somewhat short of performance goals, and generally increases in usefulness as its performance increases beyond the goals.

In addition to real-time requirements, embedded computing systems often need to restrict or minimize one or more aspects of the deployed form factor, such as volume, weight, and power consumption. For example, a computing system intended to fly on an unmanned aerial vehicle (UAV) would be constrained in all three aspects. The physical space available to the computing system may be limited, excess weight will reduce the range and endurance of the UAV, and excess power decreases the power available to other subsystems while increasing the heat that must be dissipated. As a result, the performance of embedded computing systems is frequently described not in Floating Point Operations per Second (FLOPS) but in FLOPS per Watt, FLOPS per cubic meter, or FLOPS per kilogram.

The step function utility of embedded computing systems' performance, along with the presence of form factor costs and constraints creates optimization and constraint spaces that differ from the departmental and data center systems considered in this study. For example, a computing system in a UAV that is able to exceed its time performance requirements is no more useful than one that strictly meets its requirements. For such a system, if it is possible to give up some excess computing speed in order to reduce power or space requirements, the utility of the system is improved by doing so. Holistic optimizations are necessary to maximize the overall utility of the embedded

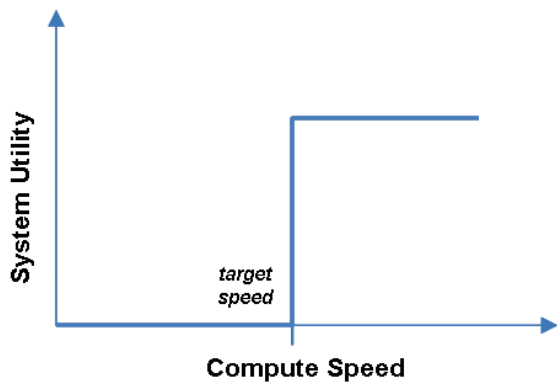


Figure A.2: Notional utility function of real-time system

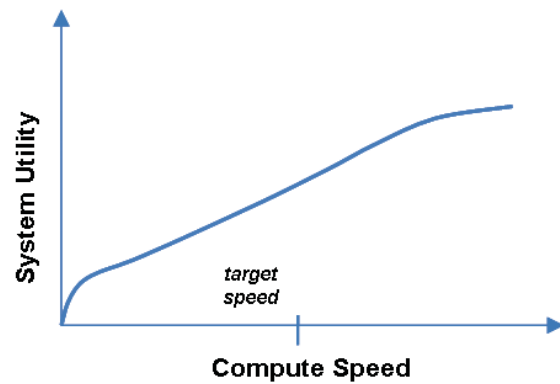


Figure A.3: Notional utility function of non-real time system

computing system. The software stack of current systems does not facilitate the automation of this optimization and constraint satisfaction. Point solutions are generated by human intervention, with coarse attempts at optimization that can, at best, locate local maxima for utility.

The ability to optimize for additional dimensions such as power, weight, and volume, as well the ability of the software stack to solve for specific constraints in each of the optimization dimensions is outside the scope of this study. Adding these capabilities to the proposed extreme scale software stack would require several additional technical improvements, including at least:

1. A formal, analyzable method for describing the constraints of elements and compositions of software, in multiple dimensions, including size, weight, power, and time.
2. A formal method to describe the utility of each dimension of the optimization space.
3. Ability at all layers of the software stack to optimize for size, weight, power, and time in arbitrary combination, as required, while simultaneously satisfying constraints in one or more of the dimensions.
4. A hardware/software co-design layer that allows the selection and configuration of computing elements to support a system.

Improvements in hardware and software for extreme scale are likely to provide direct benefit to the existing software stack for embedded systems. For example, the need for extreme scale hardware and software stacks to optimize for power as well as execution time will improve the utility of embedded computing systems that use them. Previous studies indicate that energy efficiency will be a dominating factor in the design of extreme scale computing platforms. Computing platforms designed to meet the energy efficiency needs of extreme scale systems will be attractive platforms for embedded computing systems. For such platforms, different operations may have widely different energy costs — for example, the cost of a memory fetch from the most distant intra-node memory space would expend much more memory than a floating point add of two registers. Given sufficient information about the energy cost of each operations, elements of the software stack can optimize for power as well as execution time. This ability will be necessary for extreme scale systems, and will likewise improve the utility of embedded systems.

A.2 Tools and Development Environments

A.2.1 Performance Tools

Given the complexity of Extreme Scale software, the role of performance tools must be expanded beyond showing the application programmer a set of measurements of what happened, towards synthesizing all the clues from a variety of sources and pointing the application programmer towards solutions. Extreme Scale performance tools must provide a visual presentation of what happened during program execution, and relate certain behaviors back to corresponding pieces of the code.

In support of this capability, data mining algorithms must be developed to detect anomalous behavior, and to the extent possible, this must be effective during at-scale production runs of extreme scale applications. An integration of capabilities across the software stack, as described in Chapter 7, is also needed to provide a set of “knobs” that the programmer can turn to adjust performance that can be guided by results from the performance tools.

As discussed in Chapter 7, autotuning technology should progress in support of a broader range of optimizations. Today’s autotuning technology is largely focused on locality and instruction-level parallelism, with support for multi-core code generation really just getting started. This technology must expand beyond kernels to portions of full applications, hierarchical parallelism, management of data movement, communication and parallelism tradeoffs, and performance and power tradeoffs. Research towards these expanded goals will necessarily develop domain-specific pruning heuristics and scalable empirical search techniques that make it feasible to evaluate such an expanded search space.

A.2.2 Correctness Tools

We describe correctness tools that include debuggers, analysis to proactively improve reliability, formal verification and validation.

A.2.2.1 Debuggers

The most commonly used correctness tools are debuggers, which present an interface to the application programmer to view the execution state of their application as it is running. Debuggers for parallel architectures expand the capabilities of sequential debuggers by (1) pinpointing race conditions between threads on memory accesses; and, (2) identifying communication errors in message passing code.

Debugging at extreme scale is daunting. After decades of parallel computing research, standard practice still involves fairly primitive means of examining an application’s execution state that do not scale beyond the standard 32 or fewer processors in most SMPs. With message passing codes, debugging focuses on examining communication streams and incorrect placement of barriers, but many bugs are timing sensitive and therefore intermittent. It is simply not realistic to debug each thread as a separate unit. Aggregating data for the application programmer across all threads often obscures information. Some middle ground is needed to group threads together, and to incorporate information from other correctness tools described below to focus attention on specific pieces of the computation.

New technology to increase productivity of developers during debugging of their code will rely on new sophisticated techniques and support from the software stack to make these techniques feasible at extreme scale. The remainder of the discussion on correctness tools focuses on ways to prevent or pinpoint errors through sophisticated static and dynamic analysis of the application’s execution.

A.2.2.2 Tools to Increase Reliability

Over the past decade, a large body of research and several commercial tools seek to increase the reliability of software proactively rather than in response to observed errors. The bulk of these tools look for errors in sequential code, such as memory leaks, buffer overflow, array out-of-bound errors, and similar software defects that may lead to intermittent errors. For parallel codes, analysis tools are used to detect memory race conditions, and various communication and synchronization errors. These techniques rely on a combination of static analysis and run-time testing coupled with lightweight instrumentation of the source code.

As discussed in Chapter 7, at extreme scale such techniques will be needed, not just in debugging mode, but during production runs of the application to detect errors that only appear at scale.

A.2.2.3 Formal Verification

Formal methods are a broad collection of formal specification and verification techniques to provide a rigorous verification of correctness, as an alternative to ad hoc testing. They include: (1) Formal specification methods that can elucidate critical interactions between hybrid programs; and, (2) Dynamic verification methods that can instrument a hybrid MPI/OpenMP program, and consider all its relevant execution schedules. Formal approaches are superior to traditional testing based methods in many ways. They can help verification tools automatically check for commonly committed mistakes without requiring users to create custom-made test harnesses. Many codes break only when ported to new platforms where a hitherto un-attained process interleaving suddenly manifests. Formal techniques can help identify these ‘relevant but elusive’ interleavings based on action dependence information (*e.g.*, access to common locks). Together with program instrumentation and execution control, dynamic methods can help ensure that these interleavings are considered. Last but not least, they help erect pedagogical foundations necessary for training engineers and researchers with superior skills.

MPI program verification tools (*e.g.*, [141]) are currently used to verify the absence of deadlocks, resource leaks, and communication races, and in optimization, to detect and eliminate functionally irrelevant barriers. Going into the Extreme Scale computing arena, the importance of formal methods is bound to escalate significantly. Given the sheer scale of Extreme Scale computing system designs, there will be an increased use of different programming models all the way from core-to-core communication protocols to middleware that manages multi-problem integration. Handling a plethora of such models in a seamless way, and allowing programmers to pursue efficiency while still providing multiple safety nets, are open challenges, all needing the use of formal methods. The increased propensity for faults (both hardware and software) to propagate unchecked coupled with the infeasibility of taking global checkpoints to restart failed simulation runs all calls for a judicious combination of formal methods. To be effective for Extreme Scale computing systems, we will need a combination of many formal methods technologies. In addition to static and dynamic analysis techniques mentioned above, we will need to consider formal models of compilation, memory model semantics, the correctness of compiler optimizations all the way to determining how often and what to checkpoint, defining an exception handling semantics, and last but not least, how to design adaptive power-down protocols at the hardware and software levels.

A.2.2.4 Validation

As an alternative, or in conjunction with, the formal verification techniques described above, reliability challenges at extreme scale demand new techniques for validating the correctness of a computation. Frequent non-catastrophic failures, coupled with dynamic and possibly non-deterministic

behavior, code optimized by autotuning software, and the hazards of extreme-scale parallel execution increase the likelihood that two instances of the same computation will produce different output. Conceptually, one would want to compare output of a computation or sub-computation to that of some previous and trusted execution. Combined with checkpointing, upon detecting invalid output, the computation could roll back to the previous checkpoint.

Comparing output presents numerous challenges, which is why validation is something that is currently the sole responsibility of the application programmer. Even if the amount of data to be compared is small, floating point differences across different hardware or for different but similar code demand a specification by the programmer of error tolerances. By far the biggest challenge in validating software is the sheer volume of data to be compared. Collection, storage and comparison of large volumes of output is prohibitive. Rather than full-scale comparison, simple techniques like comparing whether an output value is within a particular range, or sampling a small but representative subset of the data are examples of ways that application programmers validate their software, hard-coded into the application itself. An interesting research question is whether the extreme scale software stack can provide general support for validation, both to simplify the code and increase the use of validation to achieve more reliable software.

A.2.3 Visualization and Knowledge Discovery

While visualization is usually considered as part of understanding the results of scientific applications, visualization also plays a unique role in developing new applications, and debugging the results. Techniques from scientific visualization, which are really aimed at drawing insight from large volumes of complex data, may be brought to bear on the significant data collection and analyses involved in managing performance, power and resilience in extreme scale architectures.

The set of research challenges in this area was the subject of a Department of Energy study group which produced a report entitled, “Visualization and Knowledge Discovery: Report from the DOE/ASCR Workshop on Visual Analysis and Data Exploration at Extreme Scale”. The following key findings summarize new research areas from from Appendix A of that study:

- Mathematical Foundations: new algorithms in robust topological methods, high order tensor analysis, statistical analysis, feature detection and tracking, and uncertainty management and mitigation.
- Data Fusion: multi-modal data understanding, multi-field and multi-scale analysis and time-varying datasets.
- Exploiting Advanced Architectures and Systems: in situ processing, data access, distance visualization and end-to-end integration.
- Knowledge-Enabling Visualization and Analysis: Scientist-computer interface, collaboration, and quantitative metrics for parameter choices.

Further, related to the last item and as discussed in Scenario 6 from Chapter 7, in response to results from visualization, the application developer may wish to steer the computation in new directions.

A.2.4 Application and Execution Completion Tools

A.2.4.1 Workflow Systems

Scientific workflows capture the individual data transformations and analysis steps as well as the mechanisms to execute them. Each step in the workflow specifies a process or computation to be

executed (*e.g.*, a software program to be executed, a web service to be invoked). The steps are linked according to the data flow and dependencies among them. Workflows can capture complex analysis processes at various levels of abstraction, and also provide the provenance information necessary for scientific reproducibility, result publication, and result sharing among collaborators. By providing formalism and by supporting automation, workflows have the potential to accelerate and transform the scientific analysis process. Workflows have also become a tool capable of bringing sophisticated analysis to a broad range of users, enhancing scientific collaboration and education. Today workflows are being used in astronomy, bioinformatics, earthquake science, gravitational-wave physics, high-energy physics, and many others.

Extreme Scale workflow systems must be capable of scheduling a workflow hierarchy, formed by individual workflow tasks, entire workflows, ensembles of workflows that form an overall analysis, and workflow pools that represent computations that need to be performed at any given time. Scheduling must also cooperate with data management systems that manage data in the distributed environment at different levels of granularity from on-the-node storage associated with a node on a cluster (site), to site storage, to archival storage. Data management also needs to adhere to constraints of available disk space and policies that communities impose on managing the data life-cycle, exploring the interplay between computation and data management.

Scheduling at extreme scale cannot be statics. As failures in the execution environment occur, new failure recovery techniques need to be developed. Making sure that the computations progress in the face of failures requires sensitive monitoring systems, adaptive scheduling, computation migration, and on-demand data recovery. To minimize failures, new resource provisioning techniques need to be explored. Acquiring resources ahead of workflow execution can assure that processors are available to handle the computational tasks, can make enough disk space available to handle the data needed by and produced by computations, and reserving network bandwidth can help stage-in data when needed and stage-it out reliably and fast enough so that the storage and compute resources are available for the next set of tasks. Finally, more applications are moving into the on-demand, near-real-time performance requirements realm, thus all the computation and data management functions and capabilities need to perform efficiently and reliably.

A.2.4.2 Build systems

Build systems are used to compose source code, libraries and input data into executable applications. While building applications is relatively straightforward on commodity platforms, significant challenges arise at the extreme scale from a combination of experimental hardware, experimental software, portability issues, and simply lack of investment in tools at this scale. The following recommendations are a direct quote from the final report of a 2007 Department of Energy workshop entitled, “Workshop on Software Development Tools for Petascale Computing”:

“The current state of tools for program configuration and construction is deplorable. Applications must be built for multiple systems, including perhaps one or more petascale machines. We found that too much complexity results from multiple compilers, operating systems, libraries (and their versions). Common option sets and command-line interfaces are missing. We are concerned that the lack of shared libraries and dynamic linking capabilities on petascale systems currently in development will contribute more difficulties. We recommend consideration of new tools (make is still broken), improved tools (*e.g.*, for managing linking order), and more attention to interoperability of program build tools.”

A.2.5 Compilers

While it is obvious that compilers are needed to translate from the programming models discussed in Chapters 5 and 6, we also discuss compilers as tools, since they can and should take on additional roles in the extreme scale regime. All the distinct classification of tools mentioned above rely on compilers, coupled with the run-time environment, for program analysis, transformation, optimization and code generation, and providing an abstraction for the application programmer between the architecture and the application code.

Before looking at how compilers can support tools, we must talk about the need for a fundamental shift in how high-end application developers interact with compilers in support of all the new requirements at extreme scale. A lesson from the past two decades is that it is too ambitious to expect success from fully automatic techniques for mapping high-level application code to high-performance executables. Thus, for compiler technology to be effective, it is essential to engage the application programmer in providing domain knowledge about how tools should interact with their code. Having said this, it is still the case that application programmers have extremely limited means of interacting with compilers. Beyond compiler flags, and pragmas or programming model extensions, application programmers simply treat compilers as black boxes. To change the compiled code, they often must guess at alternative code, optimization flags, and pragmas, and empirically evaluate what produces the best result.

Compilers will continue in their role of optimizing code to exploit architectural features and improve performance. Locality and power optimizations will become a bigger focus of attention for extreme scale, as the potential for impact of such optimizations grows significantly. Given the expected dynamic and difficult-to-predict run-time behavior of extreme scale applications, optimization will increasingly become a dynamic process, and adaptive optimizations that respond to feedback from the run-time environment will grow in importance. Compilers must also provide a portion of the autotuning infrastructure described in Chapter 7 for important computational kernels, for automatically generating a set of alternative code variants and pruning the space of alternatives that are considered. Autotuning frameworks must be extended to consider portions of whole applications rather than computational kernels, consider the tradeoffs between parallelism, locality and power, and optimize more global constructs such as data organization and communication. Compilers must in some cases generate the code for the companion computations, described in Chapter 7, which are used not only to enhance performance but for many other purposes.

In terms of correctness tools, compilers should help application programmers develop correct programs through analysis to detect buffer overflow or memory violations. Compiler analysis is used in formal verification, and in generating validation code, possibly automatically. Reliability can be enhanced by supporting programming models that actually prevent certain kinds of errors. For example, languages such as Java enforce array bounds checking, type systems have been used to obtain high-level intent of the programmer, and functional languages prevent side effects across parallel threads. While such approaches are not currently in mainstream use in the HPC community, largely due to hysteresis and performance concerns, making such techniques efficient will demand new compiler support.

Compilers must support the types of interactions programmers have with the remainder of the development environment. In application completion, compilers may provide the interface for dynamic code selection and parameterized code. In general, compiler technology at extreme scale must provide mechanisms to interact with application programmers at a high-level of abstraction.

Appendix B

Definitions of Seriality, Speedup, and Scalability

Scalability has become a golden term to much of the HPC community: “this hardware is scalable; that algorithm is not scalable” but with little consistent formalism behind its use. This is particularly true when it comes to its application to system software. This discussion tries to shed some light on such a use of this term by relating it to other key terms of serialism, and speedup, and trying to develop some formal definitions. While there is nothing new in the following, we do try to relate it to exascale in a way that may help understand effects of and on software in achieving and maintaining performance.

B.1 Definitions

B.1.1 Parallelism and Concurrency

For this discussion we will distinguish between parallelism and concurrency in a rather precise way. *Parallelism* will refer to physical replication. Thus we can talk about the parallelism of function units such as floating point units (FPUs), cores, sockets (a.k.a. multi-core microprocessor chips), nodes, racks, etc.

Concurrency will refer to the overlap of operations as seen during the execution of a program, and will appear in at least three forms:

- *New concurrency*, C_{new} , will refer to the number of new operations that a program starts in whatever time units are appropriate, as in per cycle or per second. This may be the number of new instructions issued per cycle, or the number of floating point operations started per cycle. We assume that the number of operations started is equivalent to the number completed.
- *Active concurrency*, C_{active} , will refer to the total number of operations that are in some sort of execution at the same time. This reflects the pipelined nature of most function units where an operation is started in one cycle and is in computation for several more.
- *Total concurrency*, C_{total} , will refer to the total number of operations, primarily instructions, that are ready to issue but not necessarily in active computation in hardware. This would include, for example, register files in a multi-threaded core that hold currently ready to run threads that are simply waiting their turn to run on the hardware.

We note that the term “new concurrency” is most closely aligned with typical measures of performance, as in flops per cycle, or instructions per second. Also, unless otherwise specified, we will simply use the term *concurrency* to refer to new concurrency.

B.1.2 Work and Performance Metrics

For this discussion we define the *work* involved with solving a problem as the total number of basic operations (flops, instructions, etc) that need to be executed in its solution. While work and time to solution are clearly related, they are not necessarily proportional; depending on the processor’s architecture and the selected problem size, different operations (especially memory references) may take different amounts of time at different points in the computation. Thus a *performance metric* will typically be stated in terms of work per unit of time, such as flops per second, and may be stated in several forms. A *peak performance metric* is the absolute maximum amount of work that can be done in unit time by any algorithm running on some processor, regardless of the algorithm or the implementation of the algorithm in terms of a real program. A *sustained performance metric* is one that is algorithm and problem size specific, and takes into account all the delays that may exist when a real code is run on the system. It is typically the case that sustained performance metrics for real systems running the same code may vary widely with the size of the problem. Problem sizes that are, for example, “cache-resident” will exhibit far higher sustained performance numbers than ones that exceed physical memory, and require time-consuming swaps from disk.

B.1.3 Scalability

The Webster’s online dictionary defines *scalability* as either “capable of being scaled” or “capable of being easily expanded or upgraded on demand,” and as the noun *scale* as “a graded series of tests or of performances used in rating individual intelligence or achievement”. Likewise, as an adjective *to scale* is defined as “according to the proportions of an established scale of measurement”. As a verb *scaling* means “to have a specified weight on scales”.

B.2 Approximate Inter-relationships

Numerically, if the average latency of the hardware that executes a typical operation is L_{ave} , then to a first approximation:

$$C_{active} = L_{ave} * C_{new}.$$

Likewise, if the average thread can initiate instructions that correspond to on average W_{ave} of the above operations at a time, then to initiate C_{new} operations per time unit requires C_{new}/W_{ave} concurrent threads actively initiating instructions. If the longest latency for any such operation is L_{max} (typically a memory access in the hundreds or thousands of cycles), and the fraction of all operations that are such is f_{max} , then using Little’s Law the number of such operations that are active at any one time is:

$$(C_{new} * f_{max}) * L_{max}$$

This means that the number of active operations of this type that must be managed by a single active thread is:

$$(C_{new} * f_{max}) * L_{max} / (C_{new}/W_{ave}) = W_{ave} * f_{max} * L_{max}$$

Managing a long latency operation such as a remote memory reference typically requires some hardware support in a processing core such as an entry in a Load/Store queue that must be dedicated to that operation for its duration. We assume that there are Q such entries in a core. Thus, once a thread has filled all such Q entries, no further such operations can be issued, and when one is found, the core must *block* the thread, regardless of whether or not the thread still has issuable instructions. If Q is much less than the above number of outstanding operations (which is usually the case) then the only way to maintain the C_{new} level of concurrency is to have other threads ready to run. This can be done either by adding more processing cores to the system (and assume each goes idle frequently) or to *multi-thread* each core. This core or thread multiplier is thus:

$$(W_{ave} * f_{max} / Q) * L_{max}$$

As a numerical example, if we assume that $Q=4$, $W_{ave}=4$, $f_{max}=1\%$, and $L_{max}=1000$, then the above multiplier is 10.

We note that this is strictly a lower bound, since it assumes that a typical thread's program is in fact capable on average of finding at least Q independent operations that can be issued without a data dependency. We can thus bound C_{total} as:

$$C_{total} \geq (W_{ave} * f_{max} / Q) * L_{max} * C_{new}$$

B.3 Algorithmic Scalability

In high performance computing, scalability has had a long history of controversy as to its meaning and relevance [81]. We will try to be explicit by using time to solution as the basis for defining scalability, not as a general property of a system, but as relevant to only one application at a time. In this section we define $T_{A,X}(N, P)$ as the time to solution for some algorithm A when converted into a program and executed on some architecture X , where the size of the input in some units is N , and the number of “processors” allocated from X to the program execution is P . We call the combination of A and X as the *system configuration*.

For simplicity we will drop the subscripts A, X for the rest of this discussion. However it is critical to remember that all of our discussions are, in fact, relevant only to a particular system and a particular application.

To help make the following discussion more visual, we will develop a series of 3D plots of $T(N, P)$ as in Figure B.1 where the horizontal P axis is P — the number of processors, the N axis extending back into the picture is N — the size of the problem instance, and the vertical T axis is execution time in units of $T(1,1)$ (the time to solve a problem instance of minimal size 1 on a single processor).

Within such charts, a time of C is equal to an absolute execution time of $C * T(1,1)$. To make the graphs easier to explore over large variations in any direction, we assume logarithmic axes; thus the origin represents unit execution time with $N=1$ and $P=1$. In particular, $N=1$ here represents the smallest problem size for which the time complexity curve is “relevant,” and is up to the user to define.

We observe that a plot of $T(N, 1)$ represents the classical execution time curve for a single processor, and lies wholly on the NT -plane as pictured in Figure B.1. We call it the *sequential time complexity curve*. It also starts at the $(1,1,1)$ origin. As problem size increases (larger N along the N axis), the execution time increases. The notional curve in the figure is for some algorithm with greater than linear time complexity on the assumed architecture.

In the following subsections we now define two classes of scaling, depending on whether or not N is allowed to change.

September 14, 2009

Page 107

ECSS Report

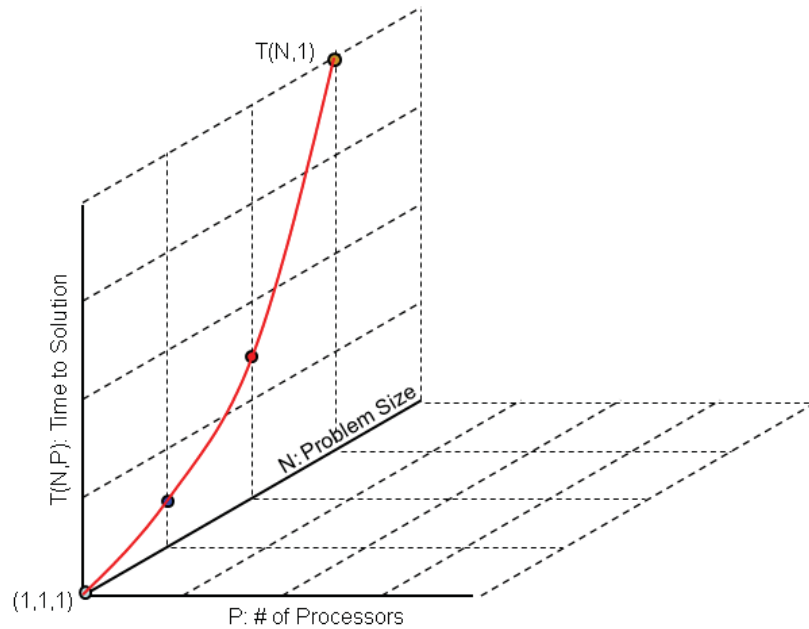


Figure B.1: Dimensions of Scalability

B.3.1 Strong Scaling

A system configuration A,X is said to exhibit *strong scaling* if, when we hold the problem size N constant, the time to solution decreases as P is allowed to increase, that is

$$T(N,P_1) > T(N,P_2) \text{ if } P_1 < P_2.$$

We say that the system exhibits *perfect strong scaling* if:

$$T(N,P) = T(N,1)/P \quad \forall P > 0.$$

In this latter case, growing the number of processors employed by some factor decreases the execution time by the same factor. Under normal circumstances, this is as good as one can expect.

Figure B.2 extends the prior Figure B.1 to reflect this concept. The curve in the PN plane is the mirror image of that in the NT plane, and represents perfect strong scaling where just enough processors (namely $P = T(N,1)$ processors) are engaged for a problem of size N to always keep the execution time at 1 unit — that of the simplest problem on one processor. The lines from the NT plane to the PN plane represent how the execution of problems of fixed size vary as the number of processors increase. These lines also assume perfect strong scaling — that is the execution time decreases as $1/P$. Finally, the lines that are in planes parallel to the NT plane reflect how using a system with some fixed number of processors works as the size of the problem changes.¹

¹Such curves obviously make sense if continued “below” the $(P=1, N=1)$ plane, as here the solution times are less than 1, *i.e.*, faster than a standard sized problem with only 1 processor.

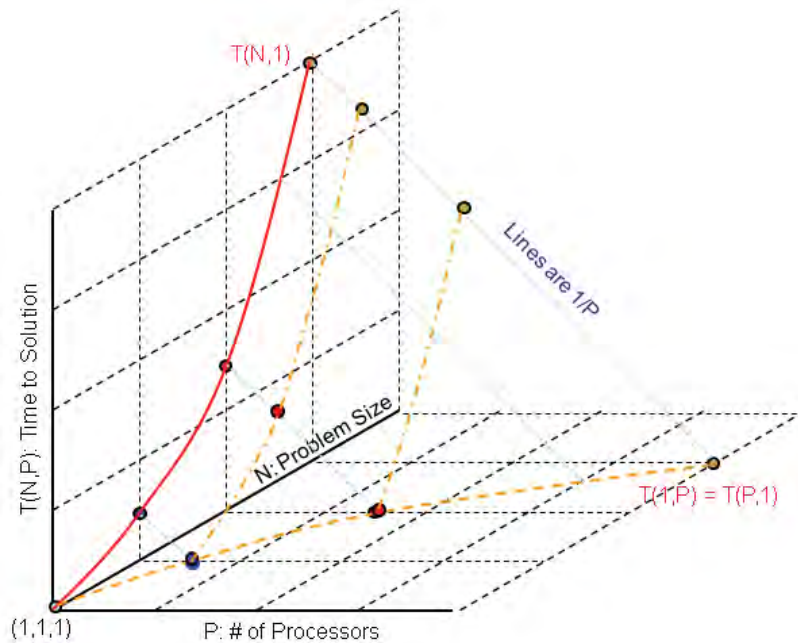


Figure B.2: Strong Scaling

The surface in Figure B.2 is important because it represents in a real sense, an upper bound for implementations of a particular algorithm. Any point on the surface represents a perfect strong speedup for a particular size of problem. As such, any point “behind” this surface is highly unlikely to be achievable (unless it demonstrates *superlinear speedup* – discussed later), without some significant variation in algorithm or architecture. Thus, it is far more likely that in the real world, strong speedups for the specified algorithm are likely to end up “in front of” this surface.

This can be bounded a bit more by adding a surface to Figure B.2 that represents “no speedup” as P increases. Figure B.3 introduces this surface as a projection perpendicular to the NT plane. Any point “in front of” this surface represents systems that “slow down” when adding additional processors (a rare but not unheard of situation). Now, any system configuration exhibiting strong scaling will correspond to a point between the two surfaces pictured.

The concept of work discussed previously also sheds some light on strong scaling. If we assume that the curve in the NT plane — the $T(N,1)$ curve — is in fact proportional to the work needed (a possibly bad assumption when N is large and memory hierarchy effects come into play), and if P processors are capable of P times the peak or sustained performance of one processor, then for a problem size of N we need $T(N,1)$ units of work. Now with P processors, for $T(N,P)$ units of time we have $T(N,P) \cdot P$ units of work available (the “peak” performance potential). However, for perfect strong scaling where $T(N,P) = T(N,1)/P$ we are using $(T(N,1)/P) \cdot P$ units of work — which is exactly all that is available. To put it another way, when perfect strong scaling occurs, the sustained performance of the processing system is independent of the size of the problem. Less than perfect scaling means that some of the processing capability is “wasted.”

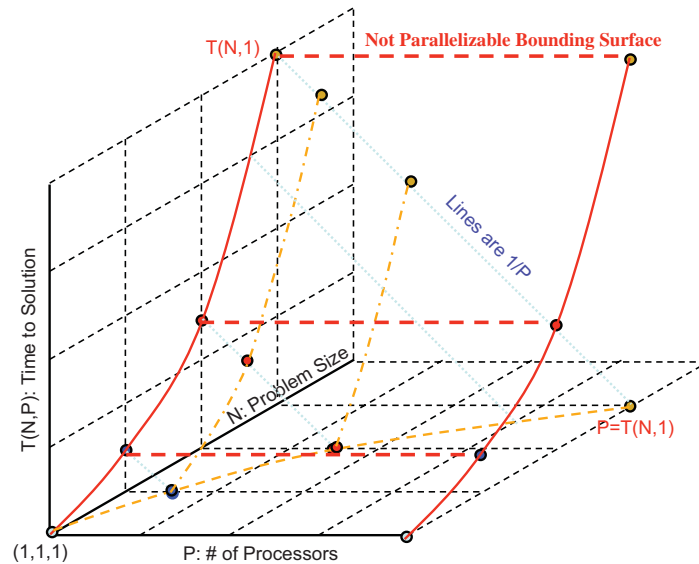


Figure B.3: Strong scaling with bounding surface

B.3.2 Weak Scaling

The other major common class of algorithm scaling is designated as *weak scaling*. While there does not appear to be a formal definition of weak scaling in the literature, the term commonly refers to the case when the amount of work performed by an application increases in proportion to the number of processors. As discussed in Section 4.2.2, the increase in work was traditionally achieved by spatial increase of the problem size but is more recently being achieved by more performing more computation per datum (“new-era” weak scaling).

The importance of weak scaling was stated in John Gustafson’s paper on “Reevaluating Amdahl’s Law” as follows [74]:

“One does not take a fixed-size problem and run it on various numbers of processors except when doing academic research; in practice, the problem size scales with the number of processors. When given a more powerful processor, the problem generally expands to make use of the increased facilities. Users have control over such things as grid resolution, number of timesteps, difference operator complexity, and other parameters that are usually adjusted to allow the program to be run in some desired amount of time. Hence, it may be most realistic to assume that run time, not problem size, is constant.”

In weak scaling, the problem size N that can be solved in constant time increases as the number of processors P increases, that is:

$$\text{For any } P_2 > P_1, \text{ there is some } N_2 > N_1 \text{ such that } T(N_2, P_2) = T(N_1, P_1).$$

It is important to realize that strong and weak scaling are *not* necessarily mutually exclusive. Each basically indicates how using additional processing can affect execution time in two different directions — strong in decreasing execution time of a fixed size problem, and weak in increasing the size of the problem without increasing time. It *is not* inconceivable to use additional processing in some way to *both* decrease execution time *and* increase problem size in some intermediate way.

September 14, 2009

Page 110

ECSS Report

B.4 Speedup

A term used with perhaps even more frequency than scalability is *speedup*. Two definitions from the web include:

- An acceleration, to go faster.
- A measure of how much faster a given program runs when executed in parallel on several processors as compared to serial execution on a single processor.

B.4.1 Defining Speedup

The latter definition leads to the typical definition of speedup as the ratio of the sequential execution time of an algorithm to its parallel execution time. In most cases, what is of real value is not the speedup for a particular algorithm at a particular level of parallelism, but an understanding of speedup as a function of parallelism P . Consequently in concert with our definition of time to solution we define $S_{A,X}(N, P)$ as the speedup for some algorithm A when converted into a program and executed on some architecture X , where the size of the input in some units is N , the number of processors allocated from X to the program execution is P , and the speedup is relative to the time for the same sized problem on a single processor. As before we will drop the subscripts.

One can raise an argument as to whether the numerator (the single processor case) should be assuming the same code as run on multiple processors, or for an optimal code written for exactly one processor (and thus avoiding all the potential overheads introduced to manage parallelism that is not used). For convention, we will assume the former, thus:

$$S(N,P) = T(N,1) / T(N,P).$$

B.4.2 Classes of Speedup

As with scalability, there are several common classes of speedup functions. *Non-parallelizable code* is where the speedup is 1 (or less) for all P . *Linear speedup* occurs when $S(N, P) \approx KP$ for some constant K and for a range of P that is of interest. Perfect linear speedup occurs when $K=1$, that is $S(N,P) = P$ for all P . While *perfect linear speedup* is the “holy grail”, several other types of speedups are actually much more common. *Logarithmic speedup* up occurs when $S(N, P) \approx P/(\log_2(P))$. *Fixed overhead speedup* occurs when $S(N,P)$ approaches some constant due to code that does not parallelize, as P approaches ∞ . *Superlinear speedup* occurs when the reduction in execution time as P increases is better than the increase in processor count. Figure B.4 places all of these speedup classes in context.

B.4.3 Fixed Overhead Speedup — Amdahl’s Law

Fixed overhead speedup typically occurs (or is believed to occur) when some fraction F of a program’s execution is purely sequential (and thus cannot be sped up with parallelism), and the rest of the program’s execution is sped up as the number of processors employed increases. If we assume perfect linear speedup for the rest of the program, then

$$T(N, P) = F * T(N, 1) + (1 - F) * T(N, 1)/P$$

Expanding $S(N,P)$ to include an extra argument F (for efficiency) yields a speedup of:

$$S(N, P, F) = T(N, 1)/(F * T(N, 1) + (1 - F) * T(N, 1)/P) = 1/(F + (1 - F)/P) \rightarrow 1/F \text{ as } P \rightarrow \infty$$

September 14, 2009

Page 111

ECSS Report

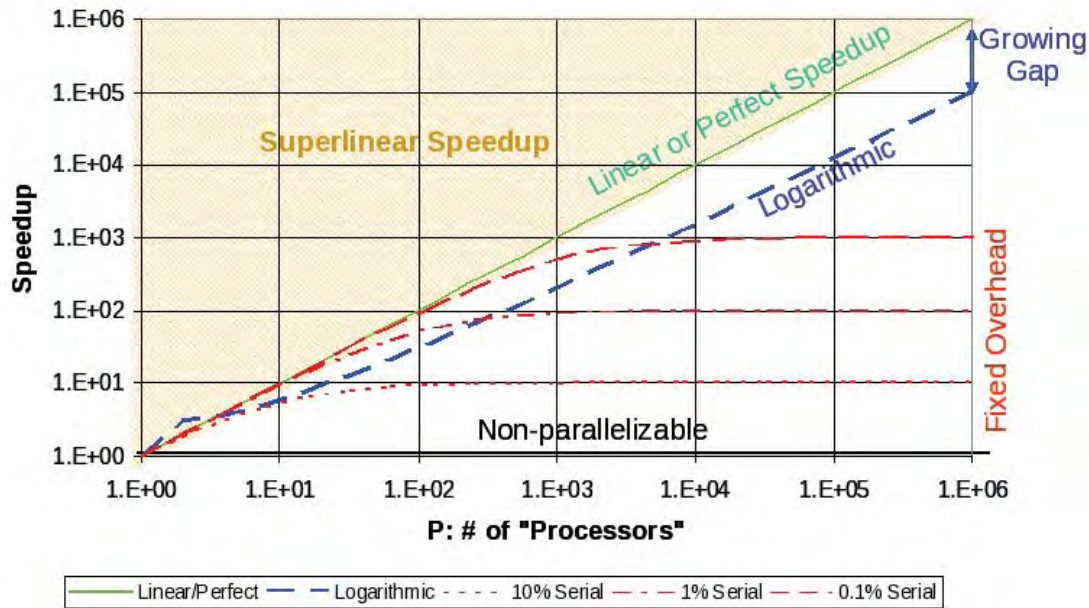


Figure B.4: Classes of speedup

We note the addition of the argument F to denote the percent of sequential time. As P goes towards infinity, this asymptotically approaches $1/F$ – the reciprocal of the fraction of sequential complexity that cannot be parallelized.

This is also known as *Amdahl's Law*. Figure B.5 diagrams this relationship for a variety of F values. As can be seen, if expected parallelism is expected to grow into the billions, the only way to get any useful speedup is to have essentially zero serialization. In this report, we often refer to the “serial part” of an application for simplicity; in practice, rather than a containing a completely serial part and a perfectly parallelizable part, an application is more likely to exhibit different scales of parallelism in different regions of code.

B.4.4 Superlinear Speedup

It is also possible, but not frequent, to have *superlinear speedup*, where the growth in speedup exceeds the growth in processors. In such cases, doubling the number of processors, for example, more than doubles the effective speedup. In terms of slopes, a superlinear curve is one where the slope is greater than one (the slope for a perfect linear speedup).

There are several example situations where such an implausible speedup might actually occur:

- Separating different “loop iterations” to different processors may in fact eliminate loop overhead.
- Increasing the number of processors may in fact increase the total effective “size” of caches near to processing cores, meaning that less latency is foisted on data memory references, and thus each processor gets relatively speaking “faster” than it was for smaller configurations.

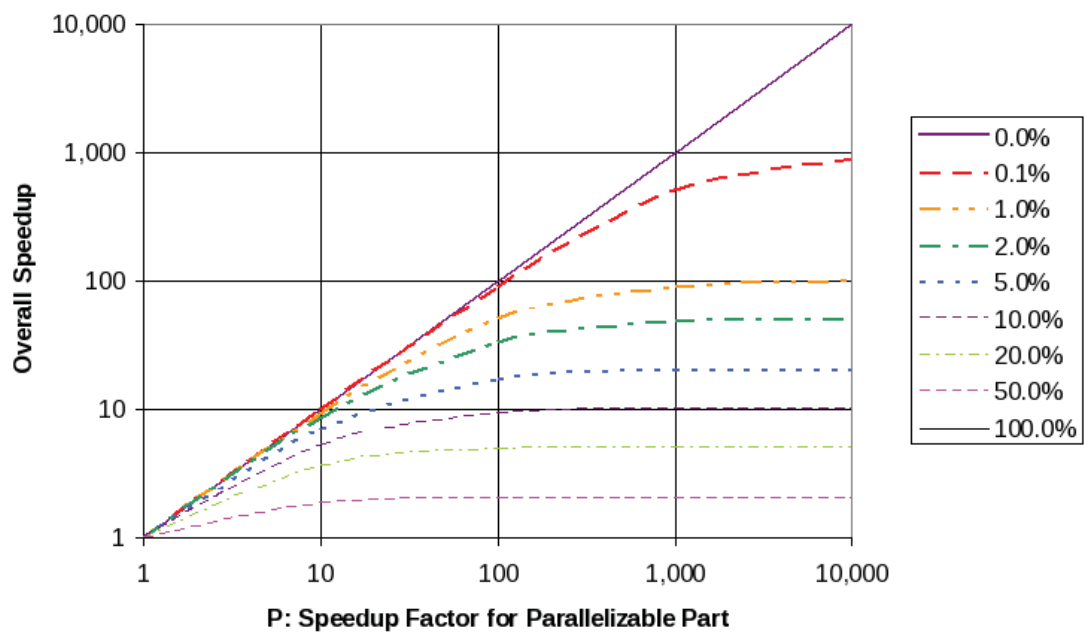


Figure B.5: Amdahl's Law

- Problems involving bounded search, such as alpha-beta search, may actually accelerate faster for larger number of processors because of an increase in breadth-first searching that reveals a tighter bound earlier than it might have in a sequential implementation, resulting in earlier cut-offs of searches across the board, and the reallocation of processors to those depth-first paths that still have promise of a better solution.

B.4.5 Speedup and Average Parallelism and Concurrency

The speedups discussed above, especially associated with Amdahl's Law, all have the implicit view that the P processors are either 100% busy during the parallel parts, or idle during the serial parts. An alternative view is that the actual degree of parallelism – how many processors are actually doing computation at any point in time, is very dynamic, and can change from moment to moment. The tree-like computations associated with parallel prefix algorithms are good examples of this — at one instant they are all busy, then only half, then $1/4^{th}$, and so on. At some time the pattern may repeat.

With this view, we may re-interpret our speedup term as an average “sustained parallelism” metric. A speedup of $S(N,P)$ for some real number of processors P is equivalent to having $S(N,P)$ processors and an algorithm that keeps them busy 100% of the time.

In addition, if we know the average pipeline depth of function units then the product with $S(N,P)$ and the number of function units per processor then gives us a realistic measure of *average concurrency* — the average number of operations that are in some state of computation at any one time. This in turn is thus related to how many *independent* operations must be extracted from the program on each and every clock cycle.

B.5 Efficiency

The above discussion of average parallelism leads naturally into a discussion on *efficiency*. Typical definitions of this term relate it to the percentage of some resource that is effectively used during execution of a program. In common usage, a typical parallel computing efficiency is defined as given some set of function units, say floating point units, over the period of computation time, what percentage actually were used in the computation. For the Linpack benchmarks used as the basis for the TOP500 list, this efficiency is often stated as R_{max} over R_{peak} , where R_{peak} is the peak floating point rate the hardware of a system is capable of, with all other constraints ignored, and R_{max} is the rate that actually was used.

Different resources may have different efficiencies during the same program, and there may in fact be a rather complex relationship between efficiencies of different resources that are “packaged” in the same subsystem. For example, we may discuss the “efficiency” of using processors as a whole in a computation, and yet that efficiency may be different when we look at the use of FPUs within the processors, or of memory bandwidth between the CPU and their local memory chips.

B.5.1 Computing Efficiency

At the top level of most discussions, the key efficiency number deals with the use of processors, and there are several different ways of determining this:

- As the ratio of total operations or instructions actually used, over the total operations or instructions possible in the algorithm's running time if all processors were always busy doing useful work = “sustained” over “peak” in our prior definitions.

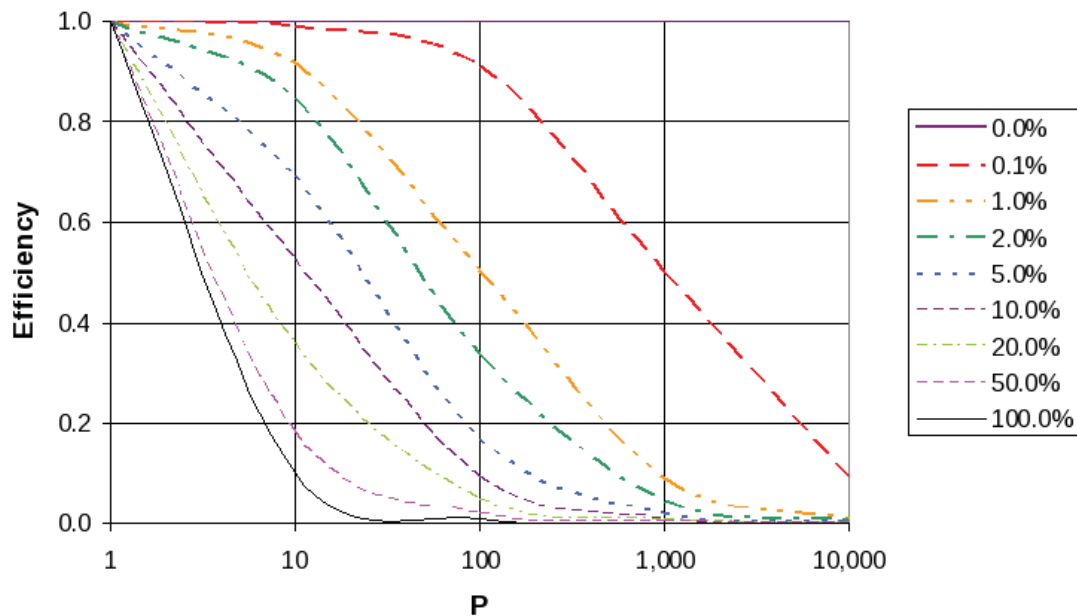


Figure B.6: Efficiency

- As the serial execution time $T(N,1)$ over the parallel execution time $T(N,P)$ times the number of processors P .
- As the average parallelism $S(N,P)$ over the peak parallelism P .

B.5.2 Efficiency and Amdahl's Law

If Amdahl's Law is a reasonable view of how some particular program behaves on some parallel computer system, then we can use the last of the above approaches to compute $E(N,P,F)$: the efficiency of use of processors for a problem of size N , processor count of P , and sequential percent F as follows:

$$E(N, P, F) = S(N, P, F)/P = (1/(F + (1 - F)/P))/P = 1/(PF + 1 - F)$$

We note that this equation is independent of the problem size. Figure B.6 graphs this relationship for a variety of F and P values. Again as can be seen, it takes a vanishingly small F value to obtain decent efficiency when there are large numbers of processors.

With this measure a 100% efficiency occurs when all the processors are doing “useful work” all the time, where “useful work” is computation at the same rate as single processor. Also, the lowest possible efficiency is $1/P$ — the equivalent work of only one processor is being done.

An interesting side bar to the above equation is to ask at what point does the efficiency drop below some arbitrary threshold. Reordering the above equation assuming a desired efficiency E , we get:

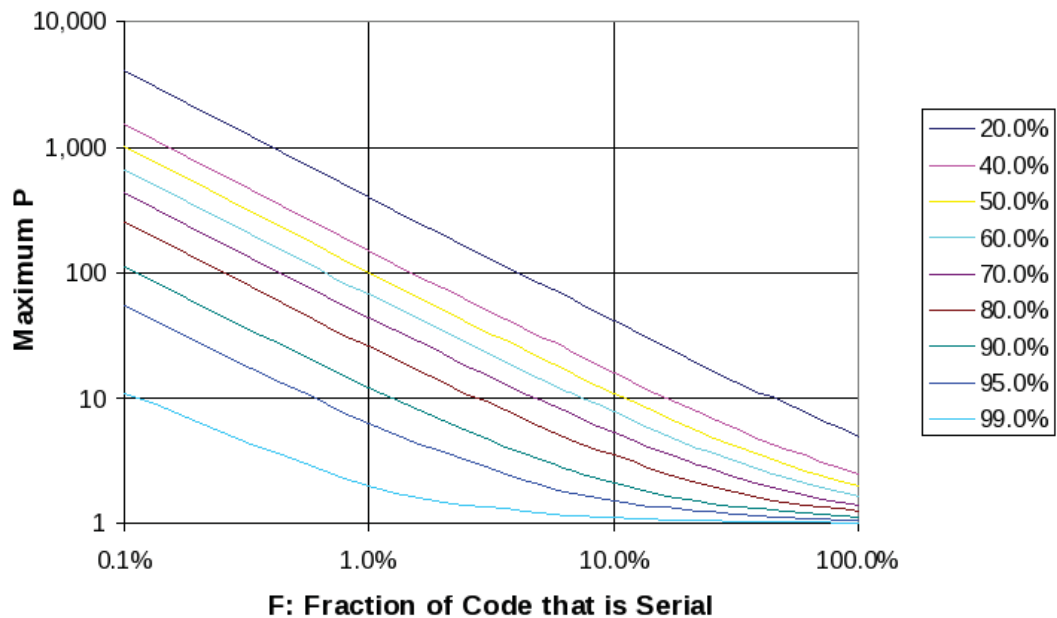


Figure B.7: Maximum usable parallelism for various efficiencies

$$P \leq (1 + EF - E)/(EF)$$

Consider, for example, a design goal of at least 50% efficiency. Using this in the above equation yields that to achieve at least 50% efficiency, we must constrain the number of processors as follows:

$$P \leq (1 + F)/F$$

Figure B.7 diagrams these bounds as a function of F for a spectrum of target efficiencies.

B.6 More Nuanced Views of Speedup

B.6.1 Gustafson's Law

Amdahl's Law assumes that for any sized problem some fixed percentage of the operations "are not parallelizable." While this may be true for some part of a problem, such as for an outermost loop overhead, it may not be true in general. In fact, it may be that as a percentage the serial overhead changes, and in particular decreases, as the size of the problem is allowed to grow. An example might be some fixed setup code at the beginning of a computation whose length is independent of problem size. Thus as the problem increases, the percent serial decreases, and parallelism is more effective.

This happens enough to be coined *Gustafson's Law*, which states that very often, if the problem size can be increased arbitrarily with sufficient parallelism then an arbitrarily large speedup can be obtained. To set this up as an equation, we assume as before that $T(N,P)$ is the time spent by a parallel processor of parallelism P solving a problem of size N . We assume also that $F(N,P)$ is the percentage of $T(N,P)$ that runs as if it is on a single processor, while $1-F(N,P)$ is the percentage that runs at the equivalent of 100% efficiency on the P parallel processors. With this we get:

$$T(N, 1) = T(N, P) * (F(N, P) + P * (1 - F(N, P)))$$

Converting this into a speedup:

$$S(N, P) = T(N, 1)/T(N, P) = F(N, P) + P * (1 - F(N, P))$$

The closer $F(N,P)$ is to zero, the smaller the P needed to reach some arbitrary speedup.

B.6.2 An Example: Fixed Overhead

As an example assume the total program execution consists of $a + bN$ instructions, where a and b are constants, and N is related to the size of the problem. We also assume that the “ bN ” part is perfectly scalable, that is with P processors it takes bN/P time. This corresponds to some algorithm of linear time complexity — a perfect weakly scalable problem. Now the parallel time is:

$$T(N, P) = a + bN/P$$

and the serial percentage is:

$$F(N, P) = a/(a + bN/P)$$

As N goes towards infinity, $F(N,P)$ approaches zero.

Plugging this into the previous speedup equation yields:

$$S(N, P) = (a + bN)/(a + bN/P)$$

Figures B.8 and B.9 diagram a case where $a = 100b$, that is the non parallelizable code is 100 times the size of the parallelizable code. As can be seen, for this problem, any desired speedup can be reached by making N (and P) big enough.

B.6.3 The Karp-Flatt Metric

The process of determining the “serial” portion of a code in advance is usually hard. An alternative is, once a program is running, to fix the problem size at N , run the same code at different degrees of parallelism, and then measure the execution time $T^*(N,P)$ (the wall clock time). We now define the effective serial fraction or the Karp-Flatt metric as $F^*(N,P)$ as the fraction of time that would still be serial if the parallelism was at 100% efficiency for the rest of the time. We can then write an equation between the measured times $T'(N,P)$ and $T'(N,1)$ and the term $F^*(N,P)$ as follows:

$$T'(N, P) = T'(N, 1) * (F^*(N, P) + (1 - F^*(N, P)) * T'(N, 1)/P)$$

or solving for $F^*(N,P)$:

$$F^*(N, P) = (P * T'(N, P) - T'(N, 1)) / (P * T'(N, 1) - T'(N, 1))$$

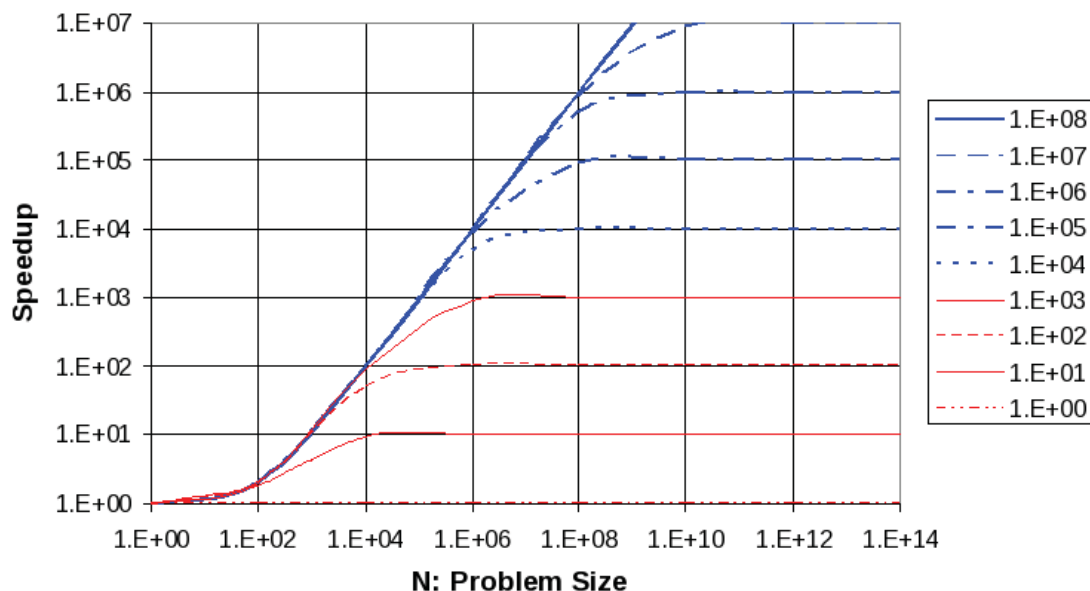


Figure B.8: Fixed overhead: Speedup as a function of N

As an example, Figure B.10 diagrams such calculations for the NAMD 2.5 code on an IBM Blue Gene computer². Figure B.11 diagrams the same data on a log curve. Note that these figures show speedup as a function of the number of processors (P) rather than the problem size (N). The latter curve in particular hints at some interesting properties as declines are interspersed with flat areas. The declines seem to occur as we move up to a drawer full of Blue Gene nodes, and then after passing beyond a board to a full rack. It would be interesting to see if further declines occur when more racks are added.

Also included on Figure B.11 is a synthetic curve of the form:

$$-0.0045 * \log_2(N) + 0.00578$$

As can be seen, this is a rather decent approximation to the observed Karp-Flatt metric up to about 1000 processors, after which it appears the metric flattens at about a 0.13% serial fraction, which, using Amdahl's Law predicts a maximum speedup of about 770.

B.7 Memory and Bandwidth Scaling

Other key parameters of real interest to any scaling discussion are memory capacity and bandwidth; both of which, if inadequate, can constrain either the size of the problems that are solvable or the rate of their solution.

²S. Kumar, G. Almasi, and L. V. Kale, "Achieving Strong Scaling On Blue Gene/L: Case Study with NAMD," 2006

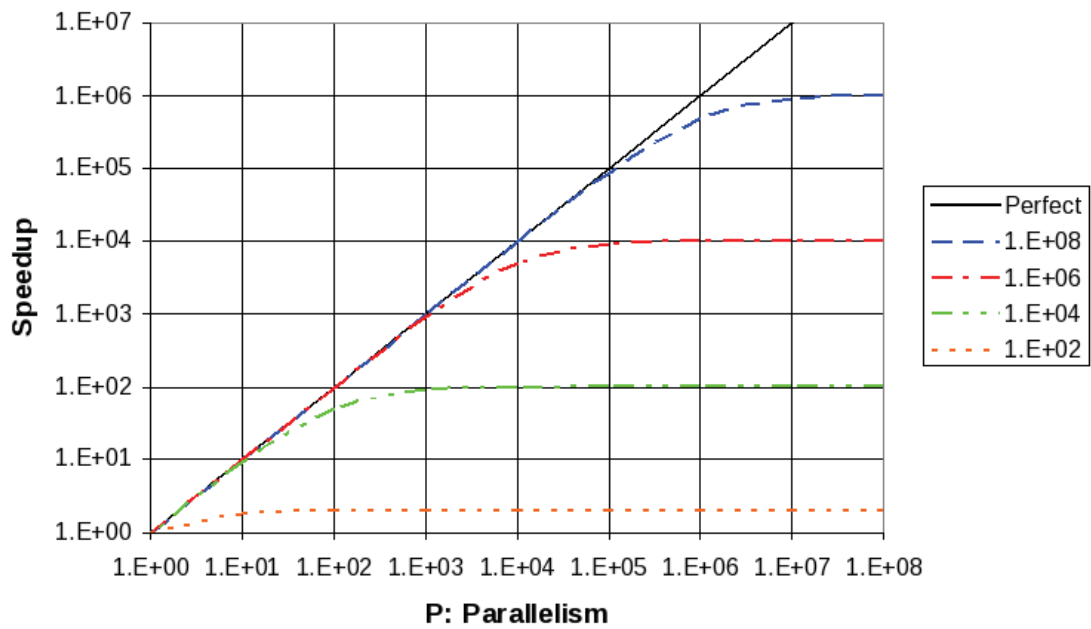


Figure B.9: Fixed overhead: Speedup as a function of P

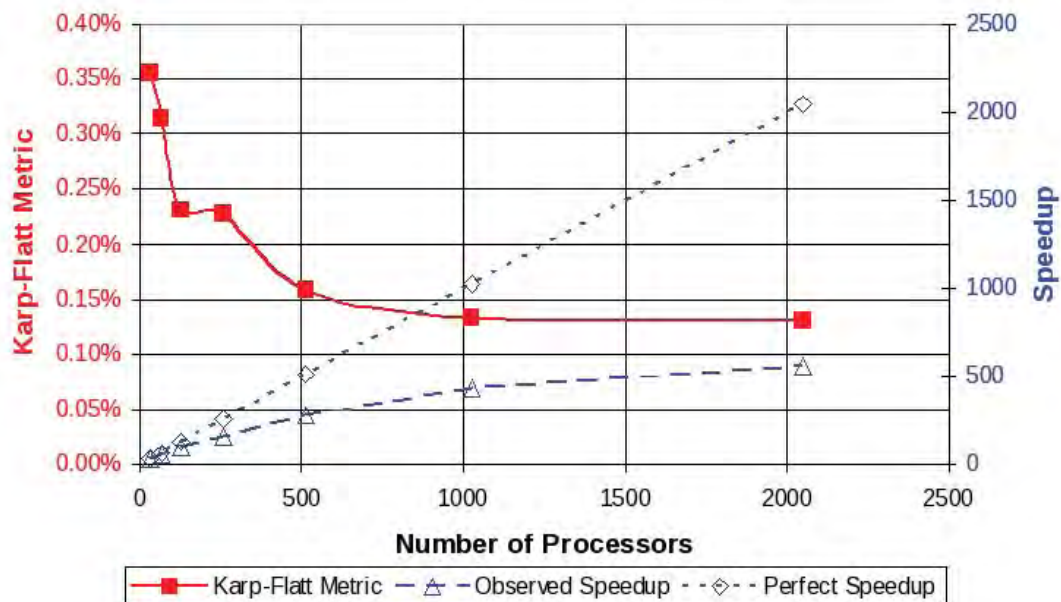


Figure B.10: Karp-Flatt metric for NAMD on a linear scale

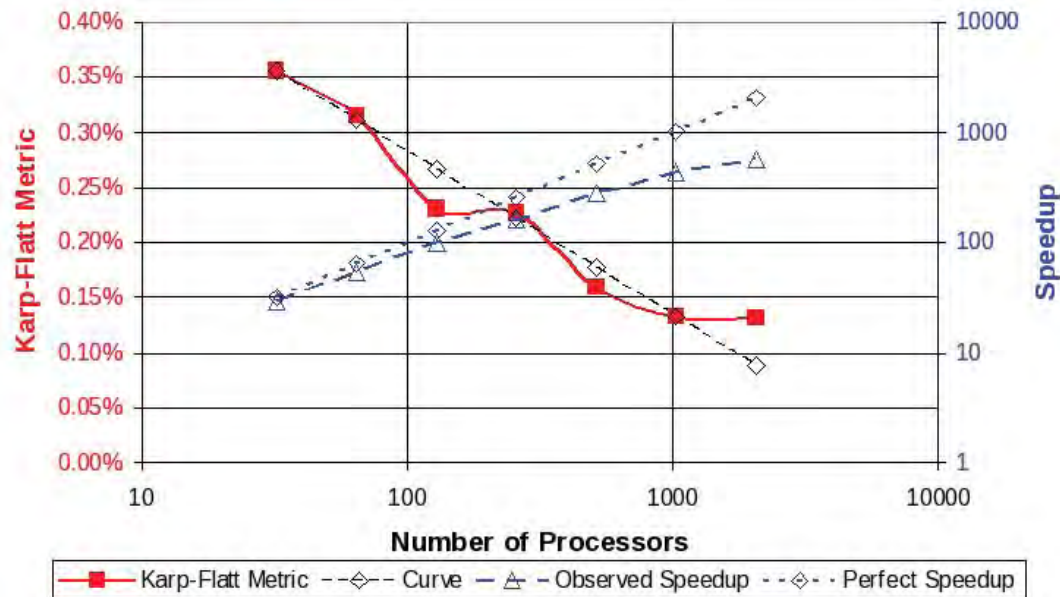


Figure B.11: Karp-Flatt metric for NAMD on a log scale

Because of the way systems are built, total memory capacity, especially for large systems, is directly proportional to the number of processors, since some fixed amount of memory is typically packaged with each processor or fixed set of processors. Available bandwidth, however, is a bit harder to extrapolate, since it is a function of both the individual nodes where the traffic is generated and of the interconnect that ties multiple nodes together. The latter, in particular, is often a function of the number of nodes and the topology of the interconnect.

On a notional level, we might expect at least the memory parameter to track the kind of scaling that the applications they support exhibit. For example, a configuration that exhibits strong scaling keeps the size of the problem constant as the number of processors increase. Thus in an ideal situation, the total required memory capacity is constant, and thus the capacity per processor might drop linearly with the number of processors.

Weak scaling assumes that the problem size increases as the number of processors increase. For perfect weak scaling, the relationship is linear; if problem size and required memory capacity is linear, then one might expect the memory capacity on a per node/processor basis to ideally stay constant as the number of processors increase.

The following subsections explore this in a bit more detail. Several candidate data structures are used as a basis for the study.

B.7.1 System Architecture and Baseline Assumptions

Understanding both of these parameters requires making some assumptions about the system architecture. For this section we assume a partitioned system where each “processor” in the prior sense has an equal-sized chunk of memory. Together we call this processor plus memory combination a

node.

We also assume mechanisms exist to allow any processor to gain access to data in other nodes. Whether this is a “pull” mechanism (as in access via load or store instructions), or “push” (as in message passing from one node to another) depends on the exact system architecture, but for now we simply assume that such access is possible, and that the cost of moving data across such node boundaries is relevant.

For memory capacity, the rule of thumb for decades has been to look at the ratio of bytes of memory capacity to flops performed; for bandwidth it is similarly the ratio of bytes per second transiting a node to the flops performed.

The magic number for decades for both numbers has been 1 to 1, but many of the most modern systems, especially at the high end, have much lower numbers. Memory ratios today, for example, run from 0.3 (commodity microprocessor-based systems) to 0.15 bytes per flop (Blue Gene-like systems). Modern GPUs are even lower, with on the order of 1 GB for a few hundred gigaflops, for ratios of around 0.002 to 0.01. The strawman design of the Exascale technology report was in the same range as the GPUs at 0.0036 bytes per flop, regardless of system size. These ratios can be a cause for concern for a) applications with computational complexity that is linearly proportional to their working set size, and b) storage systems in which the next level of the storage hierarchy is too slow for out-of-core algorithms to be practical. However, as discussed in Chapter 4, there are many applications that are not limited by a). Also, recent trends in storage technologies such as Phase Change Memory (PCM) and Flash Memory suggest that b) may not be a hard limitation either.

Bandwidth numbers are similar, but even more difficult to pin down, primarily because of the exploding trend toward multi-core chips with complex memory hierarchies. In such cases on-chip inter-core bandwidths may vary widely, as area and power for processing can be traded off for interconnect routing. Off-chip, however, is different, primarily because the number of off-chip contacts has been relatively flat for years, and the bandwidth at which these contacts can be driven is often fixed by external parameters such as cable characteristics, and the amount of power that a designer wants to spend in driving them at higher rates. The strawman exascale design from the previous report employed a tapered design, with less bandwidth available as one moved up the hierarchy (page 180, Table 7.7). Here the bytes per second per flop per second varied from 0.25 when accessing a local cache to 0.02 when accessing the (limited) local memory, to 0.006 when communicating with other nodes. Even at these highly reduced ratios, the estimated power of the upper level interconnect was 27% of the total system power, meaning that increasing such numbers can only be done at a huge power penalty. The suggested “adaptive” design was an attempt to allow at least some flexibility in boosting these ratios in cases where the processing needs are reduced.

The reason for these lower ratios is cost — both in provisioning more memory chips and in power consumed by them. Both come from both the memory chips themselves and the associated I/Os (both contacts and driver/receivers), especially when additional interface chips must be added to glue in large numbers of memory chips. This trend will continue through Exascale systems, and thus it is crucial to understand the limitations.

B.7.2 Sample Application Patterns

Any real application will have its own unique access pattern to application instance data which will change as we change the mapping of such data to node memories. However, to get some insight we define here three simple application data sets, each of size “N,” and their mappings onto P different nodes:

- *Random*: We assume here that major data structure for any run of the application is not correlatable with any placement policy, meaning that any reference made by a program in a node could be to any datum in any other node, with equal probability. Thus each node gets N/P elements of data. GUPS is an example of such an application.
- *Fully Partitioned*: In contrast to the random access mapping, here we assume that the data can be perfectly partitioned to different nodes, so that for the bulk of the application, there is no possibility of any one node accessing any other. At the end of the major processing, however, there may be some exchange of data among nodes, as in a parallel prefix computation. Again, each node gets N/P elements of data. Searching, max finding, and various forms of data clustering and data mining are possible examples of such problems.
- *Dense Multi-dimensional*: Here the data structure is a regular-shaped “cube” of dimension D, and width W of each face. Thus $N = W^D$, and each node receives a unique sub-cube of size $(W/p)^D$ where for simplicity we assume that $P = p^D$ for some p. Applications will want to access both local data within a node, and *ghost data* consisting of y “layers” of data in other nodes’ sub-cubes that are logically “at the surface” of each node’s sub-cube. Examples of such applications in one dimension might be sub-string search in a text string, in two dimensions of various matrix problems, and in three dimensions of many 3D physics models.

An additional factor in many problems with the latter multi-dimensional structure is that for weak scaling, if the growth in problem set is due to grid refinement (smaller distances between grid points), then the unit of time represented by one computation cycle also decreases, so that additional computational cycles are needed to get to the same overall period of simulated time.

B.7.3 Off-node Data

The actual amount of physical data that must be resident in each node is a function of both the data structure (as discussed above) and performance-driven considerations. In most real systems, “off-node” data is significantly further away than “on-node,” and thus there may be significant reasons to keep copies of such data in local memory. There are at least two kinds of such copied data:

- *Replicated Common Data*: This is information which is identical from one node to another, as in program code and/or lookup tables. It is typically read in at the beginning of an application’s execution, and remains unchanged for the duration. Thus for long running applications, while it may occupy memory space, it adds nothing in terms of inter-node bandwidth requirements.
- *Ghost Data*: This corresponds to the ghost surface data when the main data structure is multi-dimensional as described in Section B.7.2. Such data typically makes up a goodly chunk of inter-node communication, with either a push or a pull of the current values of all surfaces at the beginning of each iteration of the algorithm. A reasonable approximation for the size of such ghost data is; $2D * Y * (W/p)^{(D-1)}$, where Y is the number of layers.

B.7.4 Memory per Node

The minimum memory needed per node is the sum of the replicated and the non-replicated. For both the Random and Fully Partitioned data structures, there is no ghost data, so that the required memory per node should be constant for increasing P for weak scaling (problem size increases), or

decrease for increasing P for strong scaling. Whether or not the decrease in the latter case goes as $1/P$ depends on the size of any non-replicated data.

For the Multi-dimensional case, it is interesting to note that when we attempt strong scaling the percentage of data in ghost cells relative to local data grows as P increases. In fact they become equal when $p = W/(2DY)$ or $P = (W/(2DY))^D$. For the common case of 3 dimensions and one layer of ghost cells on each sub-cube surface, this corresponds to $P = N/198$, or when each node holds a cube of width 6. Further, when $P=N$ each node holds the minimum of exactly one grid point of real data, and for $D=3$ and $L=1$ has 6 ghost points around it. If we define R as the ratio of the non-replicated data with ghost to just the non-replicated data, then if we keep N constant as we increase P:

$$R = 1 + (2DY/W)*P^{1/D}$$

For $P=N/198$, $R=2$; for $P=N$, $R=7$. Again, replicated data will change these ratios

B.7.5 Off-Node

For applications with a Random data structure as defined in Section B.7.2, the bandwidth in and out of a node is a strong function only of the number of accesses made by each node into the global data, and only a weak function of the number of nodes (if accesses are truly random, then only $1/P$ of them are local, which for large P is near zero). Thus if the size of the total table and the total number of accesses to be made are held constant (*i.e.*, the problem scales strongly), then as P increases, each node holds smaller data and both generates and receives fewer accesses, but in a shorter period of time. With perfect strong scaling, the shorter time balances out the fewer references and the net bandwidth in and out of a node needs remain approximately constant.

If both the total table and the total number of accesses made grows with P (*i.e.*, the problem scales perfectly weakly), then as P increases, the number of accesses per unit time again remains approximately constant, and the bandwidth requirement is unchanged.

Applications with the Fully Partitioned model only access their local data sets until some final stage. Thus to a first approximation, their off-node bandwidth is zero regardless of how P or N changes.

Multi-dimensional applications have an entirely different off-node bandwidth scaling characteristic. In this case, the data transferred between nodes at each time step is twice the ghost data size (the ghost data from the other nodes must come in and each node must distribute part of its data to surrounding nodes as part of their ghost data). Thus, for strongly scaled configurations where the total data size is constant, and the processing is a linear function of the number of local grid points, then the off-node bandwidth is approximately twice the ghost size divided by the node's execution time, which is proportional to the number of local data points. This ratio is thus:

$$2*2D*Y*(W/p)^{(D-1)} / ((W/p)^D) = 4DYp/W$$

Since D and W are constant, the bandwidth thus actually *climbs* as the D'th root of P.

For weak scaling, the local data set size and the ghost data remains constant per node. Thus on a per computational cycle basis the bandwidth needed remains constant.

B.8 Relevance to Exascale

This section tries to draw some lessons from the above discussion in terms of the “data center” sized exascale system from the prior exascale report. We use as a baseline the clean sheet “aggressive” strawman of Section 7.3 of that report.

September 14, 2009

Page 123

ECSS Report

B.8.1 Scaling Implications

The aggressive strawman of the prior report had significantly different characteristics in many of the metrics than current generations of machines. To get some handle on what this means in terms of supporting applications, we will use the scaling relations developed above, and extrapolate from some existing systems, in particular from an XT4 and a Blue Gene/P system.

To start, we define P_{exa} as the number of cores/processors available in an exascale system (166 million in the aggressive strawman when scaled to the data center size system). Likewise P_{base} will be the equivalent number of cores in a baseline system (either the XT4 or the BG/P). Further, we assume that the extra processors in the exascale system will be used for both strong and weak scaling, and relate P_{exa} and P_{base} as follows:

$$P_{exa} = K_s * K_w * P_{base}$$

where K_s and K_w refer to how the additional processors in the exa system are used for *strong* and *weak* scaling of applications, respectively. We are free to select any combination of K_s and K_w as long as they obey the above relation and don't violate any other constraints.

In particular, if we assume we can port and then weakly scale an application that runs today on a baseline system by K_w , then this means that $K_w * P_{base}$ groups of K_s processors on the exascale system will hold an instance whose size is K_w times that running on the baseline. Further, with perfect weak scaling, using only one core per group should result in an execution time on the exascale system for the K_w scaled data set that equals the time if we had used only the original data set and P_{base} cores. This can only work if the exascale system has sufficient aggregate memory, *i.e.*,

$$\text{Memory}_{exa} \geq K_w * \text{Memory}_{base}$$

As discussed earlier, for data structures such as the multi-dimensional, it may be that this \geq relationship may require a factor of 2 or more.

Next, we assume that strong scaling will be used within each of the $K_w * P_{base}$ groups to accelerate their solution to their part of the problem. There are K_s such processors available for each such acceleration. With perfect strong scaling, the total solution time would thus drop by a factor of K_s over using only one core per group in the weakly scaled version.

Finally, since the performance characteristics of the exascale system and the baseline cores are different, we need to throw in a correction factor K_{speed} to make for an apples-to-apples time to solution comparison. As an approximation of a best case conversion factor, we compute:

$$K_{speed} = (\text{Clock}_{exa} / \text{Clock}_{base}) * (\text{Flops-per-cycle}_{exa} / \text{Flops-per-cycle}_{base})$$

Thus the best an exascale system can do is to speed up by a factor of $K_{speed} * K_s$ an application that is K_w times larger than that which will run on the baseline system, as long as:

$$\begin{aligned} K_w &\leq \text{Memory}_{exa} / \text{Memory}_{base} \\ &\text{and} \\ K_s * K_w &\leq P_{exa} / P_{base} \end{aligned}$$

One additional caveat to the above is for those cases where the data structure being expanded is a multi-dimensional data set associated with a physics problem, then the number of computational cycles needed to simulate the same period of time grows roughly with the "width" of the overall data structure. For dimensionality D , a growth in size by K_w corresponds to a growth in width of $(K_w)^{1/D}$. the effective speedup of the enlarged application to cover the same amount of simulated time must be reduced to:

Exascale Software Study

	XT4	BG/P
Maximum K_w	78	109
Maximum $K_w * K_s$	7210	2535
K_{speed}	1.19	1.82

Table B.1: Bounding parameters for scaling to aggressive exascale system.

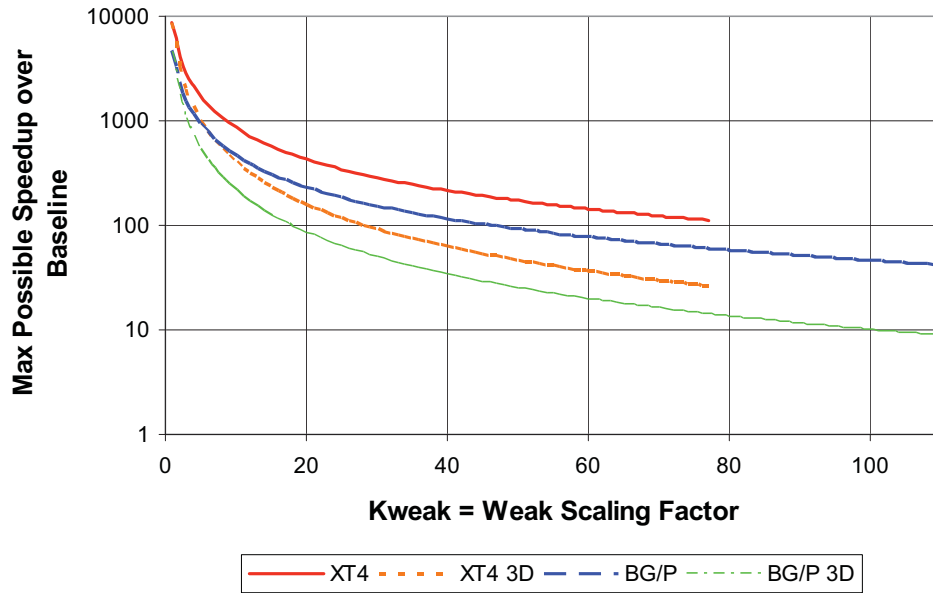


Figure B.12: Speedups as a function of weakly scaled problems.

$$K_{speed} * K_s / (K_w)^{1/D}$$

Table B.1 summarizes these constants for the two baseline systems. Figure B.12 then graphs the maximum possible speedup that an aggressive strawman exascale system might provide for applications that are scaled in size over what could be fit on a baseline system today. The lines labeled “3D” refer to cases where there are extra calculations needed to account for the refined grids.

B.8.2 Implications of Amdahl’s Law

The historical records for the TOP500 indicate an average efficiency of between 70% to 80% for the top systems over time, where efficiency here is measured in terms of flop rate, and the benchmark is Linpack. For comparison, we can ask what is the maximum serial percentage, as seen by the code run in the 166 million cores that would still yield 70% efficiency. Using the above equations, this results in a ratio of about 2.6E-9, or only about one out of every 2.6 billion instructions issued can be issued by one core when all other cores are idle. This is perhaps a million times smaller than typical serial percentages for today’s machines, and indicates a significant potential problem to be overcome.

Exascale Software Study

Sync Time (in cycles)	Minimum Time between Syncs (in cycles)
400	933
4000	9,330
40,000	93,300

Table B.2: Minimum time between sync points to maintain 70% efficiency.

B.8.3 Synchronization Points

Another way to gain insight is to ask how often can a program running on all 166 million cores go through a global synchronization before we drop to 70% efficiency. Table B.2 lists, for several different ranges of time for all threads in all cores to synchronize, what is the minimum time between sync points during which all cores must be running at 100% efficiency to maintain 70% efficiency, assuming no jitter in the completion of code that leads to the synchronization. The 400 cycle case corresponds to an architecture where direct hardware support for synchronization is provided that can sync on chip in perhaps 10 cycles, and then requires perhaps 4 traversals of the entire system. The 40,000 cycle case corresponds to a more software-implemented case where the barrier is formed as a tree of smaller barriers between groups of processors. As can, be see, the resulting times between sync points in all cases are not unreasonable.

Appendix C

CUDA as an Example Execution Model

In the last few years, as graphics processors (*GPUs*) have become more general purpose, execution models have been introduced to allow them to participate in high end computationally-intensive applications. This section overviews one such model, namely that which emerged from a line of GPUs from NVIDIA but that has been ported to other hardware from other vendors, and has in fact become in the last few months the basis for a new language standard called OPENCL¹ adopted by most major computer vendors. NVIDIA uses the name *CUDA* for *Compute Unified Device Architecture* for both the overall execution model and the programming model that accompanies it. The result is a unique combination of SPMD-like, and SIMD-like, massively multi-threaded shared memory multi-processing, that may be particularly relevant for at least some of the types of architectures that may emerge from Extreme computing technologies.

C.0.3.1 General CUDA Execution Model

The basic compute model behind CUDA assumes that data is organized into multi-dimensional arrays, and that most processing involves applying same functions to different regions of these arrays. This application of a single function to a region of data is termed a *kernel*. The execution of such functions against the smallest subset of such data is called a *thread*, and multiple threads are in general assumed to be able to execute logically independently of each other.

The specification of the mapping of different threads to different data subsets is termed an *execution configuration*. Such configurations are themselves hierarchical, with the lowest level, termed a *thread block*, representing a set of threads that are guaranteed to run more or less in tandem on the same processor at around the same time. Each thread within a block has a unique identifier, which a typical thread program then uses to associate itself with a different subset of data. Thread blocks may be up to 512 threads in size.

Sets of such thread blocks, called *thread grids*, may also be defined within a configuration, and while each thread within a block must be executed concurrently on the same processor, different blocks may execute on different processors, in arbitrary order. These grids may be multi-dimensional, and again each such block within a grid has a unique identifier that can be used to associated the threads with different data subsets.

In general, kernel invocation is asynchronous, that is multiple kernels may be executing concurrently, unless they have been linked into *streams*, where the start of one kernel is predicated

¹<http://www.khronos.org/opencl/>

upon completion of another. Multiple independent streams may be executing concurrently and interleaved in arbitrary fashions.

An explicit memory hierarchy exists, from a *global memory* that is visible to all threads, and where data is persistent throughout execution, to *local memory* (also called *shared memory*) that is allocated only for the execution period of a thread block and where only the threads started by the kernel's execution configuration may access, to *register files* that contain data that is private to individual threads. There is, however, little coherency or consistency support to memory access made by threads to these memories, especially global or shared.

C.0.3.2 Current CUDA Hardware Model

The current hardware model for NVIDIA processor chips that support CUDA assume a *host processor* of conventional design that is separate from the CUDA *device*. Both the host and the device have separate memory spaces, with only the host capable of accessing both. The device can access only its own memory. The host executes much of the overall application program control, including data transfer between host and device memories, and initiation of the highest level kernels into the device.

The device has an input queue into which kernel execution configuration requests are placed. The output of this queue schedules different thread blocks on different *Stream Multiprocessors* (SM) within the device. Each SM has within it multiple (today 8) separate *Stream Processors* (SP), each with their own register file of significant capacity (today 2048 words). Each SP is multi-threaded, meaning that it can hold the current state for dozens of separate threads, and use one of these state sets at a time to perform program computations. Each SM has a local memory (today 16384 bytes) that is accessible to all of its associated SPs. This memory can be accessed an order of magnitude faster than device global memory, and can be used as an explicitly managed cache, as well as a means of inter-SP communication. The local memory is divided into a number of banks (today 16) that allow concurrent access.

When the queue schedules a thread block, it breaks it up into fixed sized sets of threads, called *warps*, which are then executed on SMs. Today there are 32 threads in a warp. When given to an SM, the warp is distributed over the register files of the associated SPs (today 4 thread states from the warp are placed in each of the 8 SPs). The SM maintains a common program counter for all the threads in the warp, and fetches instructions for the warp one at a time. The same instruction is then given (over 4 cycles today) to all threads associated with the warp, and is then executed by each thread in the SP holding its register set.

Each thread maintains a record of whether or not it is to participate in some particular instruction so that even though the code is broadcast by the SM to all threads in a warp, each may choose individually to ignore or execute it. For example, on if-then-else blocks all threads in a warp evaluate the test condition, and then all the code associated with both the then and else paths is broadcast by the SM. Each SP decides which instructions it executes.

Loads and stores may all be executed by the SPs associated with a warp at the same time, but there is no requirement that the addresses generated by each SP be related to each other in any way. Separate hardware between the SPs and the multiple off-chip device memory ports takes the addresses generated by the SPs and *coalesces* them into patterns that maximizes the bandwidth with the memory. The address patterns that can be coalesced into concurrent memory accesses are relatively simple. *Broadcast* and adjacent accesses to global memory can generally be coalesced. Accesses to local memory can be coalesced as long as there are no accesses to multiple locations within a given bank (*i.e.*, bank conflicts).

Several warps are simultaneously assigned to each SM, which then execute in a time multiplexed

manner, allowing latency hiding of memory accesses. The GPU supports a maximum number of warps per multiprocessor (today 32) which may be reduced by insufficient number of threads available for execution, exhaustion of the SM local memory, or exhaustion of the SM register file.

This conditional execution and unpredictable execution time for individual instructions in a warp means that warps do not move in lockstep from a timing fashion. Thus each SM and its SPs support multiple warps simultaneously, and the SM will switch to issuing instructions for a different warp if not all threads associated with a prior warp have completed their instructions.

While there is no guaranteed memory consistency or operation ordering, the hardware does support instructions that allow a warp to suspend until all threads in all warps associated with a block have reached the same point.

C.0.3.3 CUDA Programming Model

For the most part, CUDA today is implemented as an extension of C, with the major differences coming in two areas. First there are *prefixes* that allow a programmer to specify whether a particular function is to be executed on the device or the host, and into which class of device memory a variable is to be allocated. Second is some additional annotation at a device function call that specifies the execution configuration of threads to be invoked: their number and structure both in grids and blocks.

Additionally, a suite of predefined variables allow a thread to determine which one it is within a block, and which block within a grid it is. Significant libraries also provide a wide range of additional capabilities.

C.0.3.4 Current Capabilities

As an example, a Tesla C1060 Computing Processor has the following characteristics²:

- 30 separate SMs on each chip, each with 8 SPs, for a total of 240 SPs.
- Each SP is capable of executing up to 3 single precision floating point operations per cycle, at up to 1.3GHz, for a peak of about 933 GFlops.
- An SM and its SPs may contain up to 8 separate thread blocks, consisting of up to 24 separate active warps, for a total of up to 768 active threads per SM.
- Up to 4 GB of GDDR3 memory may be attached, at an aggregate bandwidth of up to 102 GB/sec.

These numbers translate into a device that can support up to 23,040 concurrent threads running at an aggregate of up to about 1 Tflop single precision, with about 0.004 bytes per single precision flop of memory, and 0.1 bytes per second of bandwidth per single precision flop.

²http://www.nvidia.com/object/product_tesla_c1060_us.html

Appendix D

Extreme Scale Software Study Group Members

D.1 Committee Members

Academia & Industry	
Name	Organization
Vivek Sarkar, Chair	Rice University
Saman Amarasinghe	Massachusetts Institute of Technology
William Carlson	Institute for Defense Analyses
Andrew Chien	Intel
William Dally	Stanford University
Elmootazbellah Elnohazy	IBM
Mary Hall	University of Southern California Information Sciences Institute
Robert Harrison	Oak Ridge National Laboratory
Charles Koelbel	Rice University
David Koester	MITRE
Peter Kogge	University of Notre Dame
John Levesque	Cray
Daniel Reed	Microsoft
Robert Schreiber	Hewlett-Packard Laboratories
John Shalf	Lawrence Berkeley Laboratory
Allen Snavely	University of San Diego & San Diego Supercomputer Center
Thomas Sterling	Louisiana State University
Government and Support	
William Harrod, Organizer	DARPA
Daniel Campbell	Georgia Tech Research Institute
Kerry Hill	Air Force Research Laboratory
Jon Hiller	Science & Technology Associates
Sherman Karp	Consultant
Mark Richards	Georgia Institute of Technology
Al Scarpelli	Air Force Research Laboratory

D.2 Biographies

Saman P. Amarasinghe is an Associate Professor in the Department of Electrical Engineering and Computer Science at Massachusetts Institute of Technology and a member of the Computer Science and Artificial Intelligence Laboratory (CSAIL). Currently he leads the Commit compiler group and was the co-leader of the MIT Raw project. Under Saman's guidance, the Commit group developed the StreamIt language and compiler for the streaming domain, Superword Level Parallelism for multimedia extensions, DynamoRIO dynamic instrumentation system, Program Shepherding to protect programs against external attacks, and Convergent Scheduling and Meta Optimization that uses machine learning techniques to simplify the design and improve the quality of compiler optimization. His research interests are in discovering novel approaches to improve the performance of modern computer systems and make them more secure without unduly increasing the complexity faced by either the end users, application developers, compiler writers, or computer architects. He was also the founder of Determina Corporation, which productized Program Shepherding. Prof. Amarasinghe received his BS in Electrical Engineering and Computer Science from Cornell University in 1988, and his MSEE and Ph.D from Stanford University in 1990 and 1997, respectively.

Daniel P. Campbell is a Senior Research Engineer in the Sensors and Electromagnetic Applications Laboratory of the Georgia Tech Research Institute. Mr. Campbell's research focuses on application development infrastructure for high performance embedded computing, with an emphasis on inexpensive, commodity computing platforms. He is co-chair of the Vector Signal Image Processing Library (VSIPL) Forum, and has developed implementations of the VSIPL and VSIPL++ specifications that exploit various graphics processors for acceleration. Mr. Campbell has been involved in several programs that developed middleware and system abstractions for configurable multicore processors, including DARPA's Polymorphous Computing Architectures (PCA) program.

William W. Carlson is a member of the research staff at the IDA Center for Computing Sciences where, since 1990, his focus has been on applications and system tools for large-scale parallel and distributed computers. He also leads the UPC language effort, a consortium of industry and academic research institutions aiming to produce a unified approach to parallel C programming based on global address space methods. Dr. Carlson graduated from Worcester Polytechnic Institute in 1981 with a BS degree in Electrical Engineering. He then attended Purdue University, receiving the MSEE and Ph.D. degrees in Electrical Engineering in 1983 and 1988, respectively. From 1988 to 1990, Dr. Carlson was an Assistant Professor at the University of Wisconsin-Madison, where his work centered on performance evaluation of advanced computer architectures.

Andrew Chien is vice president of the Corporate Technology Group and director of Research for Intel Corporation. He previously served as the Science Applications International Corporation Endowed Chair Professor in the department of computer science and engineering, and the founding director of the Center for Networked Systems (CNS) at the University of California at San Diego. CNS is a university-industry alliance focused on developing technologies for robust, secure, and open networked systems. For more than 20 years, Chien has been a global leader in research and development of high-performance computing systems. His expertise includes networking, Grids, high performance clusters, distributed systems, computer architecture, high speed routing networks, compilers, and object oriented programming languages. He is a Fellow of the American Association for Advancement of Science (AAAS), Fellow of the Association for Computing Machinery

(ACM), Fellow of Institute of Electrical and Electronics Engineers (IEEE), and has published over 130 technical papers. He serves on the Board of Directors for the Computing Research Association (CRA), Advisory Board of the National Science Foundation's Computing and Information Science and Engineering (CISE) Directorate, and the Editorial Board of the Communications of the Association for Computing Machinery (CACM). From 1990 to 1998, Chien was a professor at the University of Illinois at Urbana-Champaign. During that time, he held joint appointments with both the National Center for Supercomputing Applications (NCSA) and the National Partnership for Advanced Computational Infrastructure (NPACI), working on large-scale clusters. In 1999 he co-founded Entropia, Inc., an enterprise desktop Grid company. Chien received his bachelor's in electrical engineering, master's and Ph.D. in computer science from the Massachusetts Institute of Technology.

William J. Dally is The Willard R. and Inez Kerr Bell Professor of Engineering and the Chairman of the Department of Computer Science at Stanford University. He is also co-founder, Chairman, and Chief Scientist of Stream Processors, Inc. Dr. Dally and his group have developed system architecture, network architecture, signaling, routing, and synchronization technology that can be found in most large parallel computers today. While at Bell Labs Bill contributed to the BELL-MAC32 microprocessor and designed the MARS hardware accelerator. At Caltech he designed the MOSSIM Simulation Engine and the Torus Routing Chip which pioneered wormhole routing and virtual-channel flow control. While a Professor of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology his group built the J-Machine and the M-Machine, experimental parallel computer systems that pioneered the separation of mechanisms from programming models and demonstrated very low overhead synchronization and communication mechanisms. At Stanford University his group has developed the Imagine processor, which introduced the concepts of stream processing and partitioned register organizations. Dr. Dally has worked with Cray Research and Intel to incorporate many of these innovations in commercial parallel computers, with Avici Systems to incorporate this technology into Internet routers, co-founded Velio Communications to commercialize high-speed signaling technology, and co-founded Stream Processors, Inc. to commercialize stream processor technology. He is a Fellow of the IEEE, a Fellow of the ACM, and a Fellow of the American Academy of Arts and Sciences. He has received numerous honors including the IEEE Seymour Cray Award and the ACM Maurice Wilkes award. He currently leads projects on computer architecture, network architecture, and programming systems. He has published over 200 papers in these areas, holds over 50 issued patents, and is an author of the textbooks, Digital Systems Engineering and Principles and Practices of Interconnection Networks.

Elmootazbellah N. (Mootaz) Elnozahy is a Senior Manager and Master Inventor in IBM's Research Division in Austin, Texas. He obtained a B.Sc. degree with Highest Honours in Electrical Engineering from Cairo University in 1984, and the M.S. and Ph.D. degrees in Computer Science from Rice University in 1990 and 1993, respectively. From 1993 until 1997, he was on the faculty at the School of Computer Science at Carnegie Mellon University, where he received a prestigious NSF CAREER award. In 1997, he moved to the IBM Austin Research Lab and started the Systems Software Department, which today includes over 25 researchers investigating high performance computing, low-power systems, and simulation tools. From 2005 to 2007, Mootaz joined the product division to accelerate the productization of his research project. Prior to joining IBM, he has worked on rollback-recovery, replication, and reliable distributed systems. While at IBM, he has worked on code and trace compression cc-NUMA systems, acceleration of the web site performance for the U.S. Census Bureau, blade-based servers, security of IP-based protocols, and performance tools.

He led the first two phases of the Productive, Easy-to-Use, Reliable Computing System (PERCS) project, which is IBM's effort under DARPA's HPCS initiative. Mootaz is also an Adjunct Associate Professor at the University of Texas at Austin, and has consulted with Bell Labs, Bellcore, NSF and the state of Texas. He has served on over 30 technical program committees in the areas of distributed operating systems and reliability. Mootaz's research interests include distributed systems, operating systems, computer architecture, and fault tolerance. He has published 31 refereed articles in these areas, and has been awarded 20 patents.

William Harrod joined DARPA's Information Processing Technology Office (IPTO) as a Program Manager in December of 2005. His area of interest is extreme computing, including a current focus on advanced computer architectures and system productivity, including self-monitoring and self-healing processing, Exascale computing systems, highly productive development environments and high performance, advanced compilers. He has over 20 years of algorithmic, application, and high performance processing computing experience in industry, academics and government. Prior to his DARPA employment, he was awarded a technical fellowship for the intelligence community while employed at Silicon Graphics Incorporated (SGI). Prior to this at SGI, he led technical teams developing specialized processors and advanced algorithms, and high performance software. Dr. Harrod holds a B.S. in Mathematics from Emory University, a M.S. and a Ph.D. in Mathematics from the University of Tennessee.

Mary Hall is an associate professor in the School of Computing at University of Utah. Her research focuses on compiler technology for exploiting performance-enhancing features for novel computer architectures. She received her PhD from Rice University in 1991. From 1996 to 2008, she was jointly a project leader at Information Sciences Institute and a research associate professor in the Department of Computer Science at University of Southern California. Prior to 1996, Prof. Hall held research positions at Caltech, Stanford and Rice University. Prof. Hall is currently leading the autotuning group in the DOE SciDAC Performance Engineering Research Institute. Previously, she was principal investigator on DIVA (Data-IntensiVe Architecture), a system architecture project that utilizes processing logic internal to memory chips as smart memory co-processors, and on DEFECTO (Design Environment for Adaptive Computing), an end-to-end design environment for FPGA-based computing environments. Prof. Hall has served on over 40 program committees in compilers and their interaction with architecture, parallel computing, and embedded and reconfigurable computing, including 2010 program chair for the ACM Principles and Practice of Parallel Programming and 2009 program chair of the Code Generation and Optimization Conference, and workshop co-chair for SC08, among others. She has authored over 70 papers in these areas. She is active in the ACM, serving as chair of the ACM History Committee, and previously on the ACM Health of Conferences Committee. She also participates in outreach programs to encourage the participation of women in computer science.

Robert Harrison Robert J. Harrison holds a joint appointment between Oak Ridge National Laboratory (ORNL) and the University of Tennessee, Knoxville. At the university, he is a professor in the chemistry department. At ORNL he is a corporate fellow and leader of the Computational Chemical Sciences Group in the Computer Science and Mathematics Division. He has many publications in peer-reviewed journals in the areas of theoretical and computational chemistry, and high-performance computing. His undergraduate (1981) and post-graduate (1984) degrees were obtained at Cambridge University, England. Subsequently, he worked as a postdoctoral research fellow at the Quantum Theory Project, Univ. Florida, and the Daresebury Laboratory, England,

before joining the staff of the theoretical chemistry group at Argonne National Laboratory in 1988. In 1992, he moved to the Environmental Molecular Sciences Laboratory of Pacific Northwest National Laboratory, conducting research in theoretical chemistry and leading the development of NWChem, a computational chemistry code for massively parallel computers. In August 2002, he started the joint faculty appointment with UT/ORNL. In addition to his DOE Scientific Discovery through Advanced Computing (SciDAC) research into efficient and accurate calculations on large systems, he has been pursuing applications in molecular electronics and chemistry at the nanoscale. In 1999, the NWChem team received an R&D Magazine R&D100 award, and, in 2002, he received the IEEE Computer Society Sydney Fernbach award.

Kerry L. Hill is a Senior Electronics Engineer with the Advanced Sensor Components Branch, Sensors Directorate, Air Force Research Laboratory, Wright-Patterson AFB OH. Ms. Hill has 27 years experience in advanced computing hardware and software technologies. Her current research interests include advanced digital processor architectures, real-time embedded computing, and re-configurable computing. Ms. Hill worked computer resource acquisition technical management for both the F-117 and F-15 System Program Offices before joining the Air Force Research Laboratory in 1993. Ms. Hill has provided technical support to several DARPA programs including Adaptive Computing Systems, Power Aware Computing and Communications, Polymorphous Computing Architectures, and Architectures for Cognitive Information Processing.

Jon C. Hiller is a Senior Program Manager at Science and Technology Associates, Inc. Mr. Hiller has provided technical support to a number of DARPA programs, and specifically computing architecture research and development. This has included the Polymorphous Computing Architectures, Architectures for Cognitive Information Processing, Power Aware Computing and Communications, Data Intensive Systems, Adaptive Computing Systems, and Embedded High Performance Computing Programs. Previously in support of DARPA and the services, Mr. Hiller's activities included computer architecture, embedded processing application, autonomous vehicle, and weapons systems research and development. Prior to his support of DARPA, Mr. Hiller worked at General Electric's Military Electronic Systems Operation, Electronic Systems Division in the areas of GaAs and CMOS circuit design, computing architecture, and digital and analog design and at Honeywell's Electro-Optics Center in the areas of digital and analog design. Mr. Hiller has a BS from the University of Rochester and a MS from Syracuse University in Electrical Engineering.

Sherman Karp has been a consultant to the Defense Research Projects Agency (DARPA) for the past 21 years and has worked on a variety of projects including the High Productivity Computing System (HPCS) program. Before that he was the Chief Scientist for the Strategic Technology Office (STO) of DARPA. At DARPA he did pioneering work in Low Contrast (sub-visibility) Image enhancement and Multi-Spectral Processing. He also worked in the area of fault tolerant spaceborne processors. Before moving to DARPA, he worked at the Naval Ocean Systems Center where he conceived the concept for Blue-Green laser communications from satellite to submarine through clouds and water, and directed the initial proof of principle experiment and system design. He authored two seminal papers on this topic. For this work he was named the NOSC Scientist of the Year (1976), and was elected to the rank of Fellow in the IEEE. He is currently a Life Fellow. He has co-authored four books and two Special Issues of the IEEE. He was awarded the Secretary of Defense Medal for Meritorious Civilian Service, and is a Fellow of the Washington Academy of Science, where he won the Engineering Sciences Award. He was also a member of the Editorial Board of the IEEE Proceedings, the IEEE FCC Liaison Committee, the DC Area

IEEE Fellows Nomination Committee, the IEEE Communications Society Technical Committee on Communication Theory, on which he served as Chairman from 1979-1984, and was a member of the Fellows Nominating Committee. He is also a member of Tau Beta Pi, Eta Kappa Nu, Sigma Xi and the Cosmos Club.

David Koester received his Masters in Applied Statistics (MAS) from The Ohio State University in 1978 and his doctorate from Syracuse University in 1996 under the guidance of Dr. Geoffrey Fox and Dr. Sanjay Ranka. He joined the MITRE Corporation at the MITRE-Rome site in 1978 and continues to work from that office which is co-located with the Air Force Research Laboratory (AFRL) Site Rome, NY. Dr. Koester's present areas of interest in High End Computing (HEC) technologies include understanding the high-level mappings of applications to computing architectures and metrics to evaluate system performance and productivity.

Peter M. Kogge is currently the Associate Dean for research for the College of Engineering, the Ted McCourtney Chair in Computer Science and Engineering, and a Concurrent Professor of Electrical Engineering at the University of Notre Dame, Notre Dame, Indiana. From 1968 until 1994, he was with IBM's Federal Systems Division in Owego, NY, where he was appointed an IBM Fellow in 1993. In 1977 he was a Visiting Professor in the ECE Dept. at the University of Massachusetts, Amherst, MA, and from 1977 through 1994, he was also an Adjunct Professor of Computer Science at the State University of New York at Binghamton. He has been a Distinguished Visiting Scientist at the Center for Integrated Space Microsystems at JPL, and the Research Thrust Leader for Architecture in Notre Dame's Center for Nano Science and Technology. For the 2000-2001 academic year he was also the Interim Schubmehl-Prein Chairman of the CSE Dept. at Notre Dame. His research areas include advanced VLSI and nano technologies, non von Neumann models of programming and execution, parallel algorithms and applications, and their impact on massively parallel computer architecture. Since the late 1980s' this has focused on scalable single VLSI chip designs integrating both dense memory and logic into "Processing In Memory" (PIM) architectures, efficient execution models to support them, and scaling multiple chips to complete systems, for a range of real system applications, from highly scalable deep space exploration to trans-petaflops level supercomputing. This has included the world's first true multi-core chip, EXECUBE, that in the early 1990s integrated 4 Mbits of DRAM with over 100K gates of logic to support a complete 8 way binary hypercube parallel processor which could run in both SIMD and MIMD modes. Prior parallel machines included the IBM 3838 Array Processor which for a time was the fastest single precision floating point processor marketed by IBM, and the Space Shuttle Input/Output Processor which probably represents the first true parallel processor to fly in space, and one of the earliest examples of multi-threaded architectures. His Ph.D. thesis on the parallel solution of recurrence equations was one of the early works on what is now called parallel prefix, and applications of those results are still acknowledged as defining the fastest possible implementations of circuits such as adders with limited fan-in blocks (known as the Kogge-Stone adder). More recent work is investigating how PIM-like ideas may port into quantum cellular array (QCA) and other nanotechnology logic, where instead of "Processing-In-Memory" we have opportunities for "Processing-In-Wire" and similar paradigm shifts.

John Levesque is the Director of Cray's Supercomputing Center of Excellence based at Oak Ridge National Laboratory (ORNL). The group is tasked with assisting the DoE Office of Science Researchers in porting and scaling their applications to the Petascale systems based at ORNL. Mr. Levesque is in the Chief Technology Office of Cray Inc, responsible for Application awareness

throughout the company. Prior to joining Cray, he was the Director of the Advanced Computing Technology Center based in IBM Research. Mr. Levesque started his career in high performance computing 40 years ago as an application developer at Sandia Laboratory and Air Force Weapons Laboratory in Albuquerque, New Mexico. He joined R&D Associates in Los Angeles, CA in 1972 where he ran a DARPA project to monitor ILLIAC IV code development efforts. Until joining IBM in 1998, he ran software development groups at Pacific Sierra Research and Applied Parallel Research. These groups pioneered parallel programming tools using “whole program” analysis in the VAST and FORGE software. Mr. Levesque co-authored a book “Guidebook to Fortran on Supercomputers” in 1981 and is currently working on a similar book addressing effective programming for multi-core architectures. Mr. Levesque received his Master Degree in Mathematics at the University of New Mexico in 1972.

Daniel Reed is Microsoft’s Scalable and Multicore Computing Strategist, responsible for re-envisioning the mega-data center of the future. Previously, he was the Chancellor’s Eminent Professor at UNC Chapel Hill, as well as the Director of the Renaissance Computing Institute (RENCI) and the Chancellor’s Senior Advisor for Strategy and Innovation for UNC Chapel Hill. Dr. Reed has served as a member of the U.S. President’s Council of Advisors on Science and Technology (PCAST) and as a member of the President’s Information Technology Advisory Committee (PITAC). He recently chaired a review of the U.S. networking and IT research portfolio, and he recently completed a term as chair of the board of directors of the Computing Research Association. He was previously Head of the Department of Computer Science at the University of Illinois at Urbana-Champaign (UIUC). He has also been Director of the National Center for Supercomputing Applications (NCSA) at UIUC, where he also led National Computational Science Alliance. He was also one of the principal investigators and chief architect for the NSF TeraGrid. He received his PhD in computer science in 1983 from Purdue University.

Mark A. Richards is a Principal Research Engineer and Adjunct Professor in the School of Electrical and Computer Engineering, Georgia Institute of Technology. From 1988 to 2001, Dr. Richards held a series of technical management positions at the Georgia Tech Research Institute, culminating as Chief of the Radar Systems Division of GTRI’s Sensors and Electromagnetic Applications Laboratory. From 1993 to 1995, he served as a Program Manager for the Defense Advanced Research Projects Agency’s (DARPA) Rapid Prototyping of Application Specific Signal Processors (RASSP) program, which developed new computer-aided design (CAD) tools, processor architectures, and design and manufacturing methodologies for embedded signal processors. Since the mid-1990s, he has been involved in a series of programs in high performance embedded computing, including the efforts to develop the Vector, Signal, and Image Processing Library (VSIPL) and VSIPL++ specifications and the Stream Virtual Machine (SVM) middleware developed under DARPA’s Polymorphous Computing Architectures (PCA) program. Dr. Richards is the author of the text Fundamentals of Radar Signal Processing (McGraw-Hill, 2005).

Vivek Sarkar (Chair) is the E.D. Butcher Professor of Computer Science at Rice University. He conducts research in programming languages, compiler optimizations and runtime systems for parallel and high performance computer systems, and currently leads the Habanero Multicore Software Research project at Rice University. Prior to joining Rice, he was Senior Manager of Programming Technologies at IBM Research. His responsibilities at IBM included leading IBM’s research efforts in programming model, tools, and productivity in the PERCS project during 2002- 2007 as part of the DARPA High Productivity Computing System program. His past projects include the X10

programming language, the Jikes Research Virtual Machine for the Java language, the ASTI optimizer used in IBM's XL Fortran product compilers, the PTRAN automatic parallelization system, and profile-directed partitioning and scheduling of Sisal programs. Vivek became a member of the IBM Academy of Technology in 1995, the E.D. Butcher Professor of Computer Science at Rice University in 2007, and was inducted as an ACM Fellow in 2008. He holds a B.Tech. degree from the Indian Institute of Technology, Kanpur, an M.S. degree from University of Wisconsin-Madison, and a Ph.D. from Stanford University. In 1997, he was on sabbatical as a visiting associate professor at MIT, where he was a founding member of the MIT RAW multicore project.

Alfred J. Scarpelli is a Senior Electronics Engineer with the Advanced Sensor Components Branch, Sensors Directorate, Air Force Research Laboratory, Wright-Patterson AFB OH. His current research areas include advanced digital processor architectures, real-time embedded computing, and reconfigurable computing. Mr. Scarpelli has 33 years research experience in computer architectures and computer software. In the 1970's, he conducted benchmarking to support development of the MIL-STD-1750 instruction set architecture, and test and evaluation work for the DoD standard Ada language development. In the 1980's, he was involved with the DoD VHSIC program, and advanced digital signal processor development, a precursor to the F-22 Common Integrated Processor. In the 1990's, his research focused on the DARPA Pilot's Associate, the development of an embedded, artificial intelligence processor powered by an Associative Memory CoProcessor, real-time embedded software schedulability techniques, VHDL compilation and simulation tools, and application partitioning tools for reconfigurable computing platforms. Since 1997, he has provided technical support to multiple DARPA programs such as Adaptive Computing Systems, Polymorphous Computing Architectures, Architectures for Cognitive Information Processing, Networked Embedded Systems Technology, Mission Specific Processing, and Foliage Penetration. He holds a B.S. degree in Computer Science from the University of Dayton (1979) and an M.S. degree in Computer Engineering from Wright State University (1987).

Rob Schreiber is a Distinguished Technologist in and Assistant Director of the Exascale Computing Lab at Hewlett Packard Laboratories. Dr. Schreiber received an AB in mathematics from Cornell in 1972 and a PhD in Computer Science from Yale in 1977. He is known for research in sequential and parallel algorithms for matrix computation and compiler optimization for parallel languages. He was a professor of Computer Science at Stanford and at RPI and was chief scientist of the Saxpy Computer company. He was a co-developer of the sparse matrix extension of Matlab, and a leading designer of the High Performance Fortran programming language. He was one of the developers of the NAS parallel benchmarks. At HP, Rob helped lead the PICO Project, which developed a system for embedded processor synthesis from high-level specifications. In 2007 he was named as a Distinguished Scientist by the Association for Computing Machinery.

John Shalf is with the National Energy Research Scientific Computing Center of the Lawrence Berkeley National Laboratory. His background is in electrical engineering. He spent time in graduate school at Virginia Tech working on a C-compiler for the SPLASH-2 FPGA-based computing system, and at Spatial Positioning Systems Inc. (now ArcSecond) he worked on embedded computer systems. John first got started in HPC at the National Center for Supercomputing Applications (NCSA) in 1994, where he provided software engineering support for a number of scientific applications groups. While working for the General Relativity Group at the Albert Einstein Institute in Potsdam Germany, he helped develop the first implementation of the Cactus Computational Toolkit, which is used for numerical solutions to Einstein's equations for General Relativity and

which enables modeling of black holes, neutron stars, and boson stars. John joined Berkeley Lab in 2000 where he co-authored the "View from Berkeley" report with David Patterson et. al. at UC Berkeley, which discussed the future research challenges of multicore computing. He currently leads the Science Driven System Architecture group at the National Energy Research Supercomputing Center and leads the *Green Flash* project at LBL to develop energy-efficient computer architectures.

Allan E. Snavely is an Adjunct Assistant Professor in the University of California at San Diego's Department of Computer Science and is founding director of the Performance Modeling and Characterization (PMaC) Laboratory at the San Diego Supercomputer Center. He is a noted expert in high performance computing (HPC). He has published more than 50 papers on this subject, has presented numerous invited talks including briefing U.S. congressional staff on the importance of the field to economic competitiveness, was a finalist for the Gordon Bell Prize 2007 in recognition for outstanding achievement in HPC applications, and is primary investigator (PI) on several federal research grants. Notably, he is PI of the Cyberinfrastructure Evaluation Center supported by National Science Foundation, and Co-PI in charge of the performance modeling thrust for PERI (the Performance Evaluation Research Institute), a Department of Energy SciDAC2 institute.

Thomas Sterling is the Arnaud and Edwards Professor of Computer Science at Louisiana State University and a member of the Faculty of the Center for Computation and Technology. Dr. Sterling is also a Faculty Associate at the Center for Computation and Technology at California Institute of Technology and a Distinguished Visiting Scientist at Oak Ridge National Laboratory. Sterling is an expert in the field of parallel computer system architecture and parallel programming methods. Dr. Sterling led the Beowulf Project that performed seminal pathfinding research establishing commodity cluster computing as a viable high performance computing approach. He led the Federally sponsored HTMT project that conducted the first Peta ops scale design point study that combined advanced technologies and parallel architecture exploration as part of the national peta ops initiative. His current research directions are the ParalleX execution model and processor in memory architecture for directed graph based applications. He is a winner of the Gordon Bell Prize, co-author of five books, and holds six patents.

Appendix E

Extreme Scale Software Study Meetings, Speakers, and Guests

Meeting # 1 (Kickoff Meeting)

June 17, 2008, Boston, MA

Host: Massachusetts Institute of Technology

Committee members present

Dan Campbell, Andrew Chien, Bill Dally, Mootaz Elnohazy, Mary Hall, Robert Harrison, Bill Harrod, Jon Hiller, Sherman Karp, David Koester, John Levesque, John Shalf, Vivek Sarkar, Allan Snively

Visitors

- Anant Agarwal, Massachusetts Institute of Technology
- Guy Steele, Sun Microsystems

Presentations

- “Introduction” — Vivek Sarkar:
- “Project Overview” — Bill Harrod
- “Software Challenges and Approaches for 1000 Cores: the CSAIL Angstrom Project” — Anant Agarwal
- “Breaking Sequential Habits of Thought” — Guy Steele

Meeting # 2

July 9, 2008, Atlanta, GA

Host: Georgia Institute of Technology

Committee members present

Saman Amarasinghe, Bill Carlson, Dan Campbell, Andrew Chien, Bill Dally, Mootaz Elnohazy, Bill Harrod, Jon Hiller, Sherman Karp, Peter Kogge, John Levesque, Mark Richards, Vivek Sarkar, Allan Snively

Visitors

- David Bader, Georgia Institute of Technology
- Guy Steele, Sun Microsystems

Presentations

- “Status Update” — Vivek Sarkar
- “Concurrency at Exascale” — Peter Kogge
- “Exascale Analytics in Biology, Social Networks, and Security” - David Bader

Meeting # 3

July 31, 2008, Argonne, IL

Host: Argonne National Laboratory

Committee members present

Saman Amarasinghe, Dan Campbell, Bill Dally, Thomas Dunning, Mootaz Elnohazy, Mary Hall, Robert Harrison, Bill Harrod, Jon Hiller, Sherman Karp, Charles Koelbel, John Levesque, Vivek Sarkar, Allan Snively

Visitors

- Peter Beckman, Argonne National Laboratory
- William Gropp, University of Illinois at Urbana-Champaign
- Rusty Lusk, Argonne National Laboratory
- Arvind Mithal, Massachusetts Institute of Technology
- Rob Pennington, University of Illinois at Urbana-Champaign
- Rob Schreiber, Hewlett-Packard Laboratories
- Marc Snir, University of Illinois at Urbana-Champaign
- Guy Steele, Sun Microsystems

Presentations

- “Status Update” — Vivek Sarkar
- “Programming models for Petascale Computing, and beyond” — Marc Snir
- “Scalability Challenges in System Software” — Pete Beckman
- “Reliability in Large-Scale Systems (DARPA Exascale Computing Resiliency Study)” — Mootaz Elnozahy

Meeting # 4

August 12, 2008, Houston, TX

Host: Rice University

Committee members present

Saman Amarasinghe (via teleconference), Dan Campbell, Mootaz Elnohazy, Mary Hall, Bill Harrod, Jon Hiller, Sherman Karp, Charles Koelbel, David Koester, Peter Kogge, John Levesque, Mark Richards, Vivek Sarkar, Allan Snively, Tom Sterling

Visitors

- Jack Dongarra, University of Tennessee at Knoxville
- John Mellor-Crummey, Rice University
- Krishna Palem, Rice University
- Rob Schreiber, Hewlett-Packard Laboratories

Presentations

- “Status Update” — Vivek Sarkar
- “Scheduling for Numerical Linear Algebra Library at Scale” — Jack Dongarra
- “Tool Challenges for Exascale Computing” — John Mellor-Crummey
- “Compiler Optimizations for Power Aware Computing” — Krishna Palem

Meeting # 5

September 16, 2008, Stanford, CA

Host: Stanford University

Committee members present

Saman Amarasinghe, Dan Campbell, Andrew Chien, Bill Dally, Mootaz Elnohazy, Mary Hall (via teleconference), Robert Harrison, Bill Harrod, Jon Hiller, Sherman Karp, Charles Koelbel (via teleconference), David Koester, Mark Richards, Vivek Sarkar (via teleconference), John Shalf, Allan Snively

September 14, 2009

Page 141

ECSS Report

Visitors

- Ron Brightwell, Sandia National Laboratories
- G. R. Gao, University of Delaware
- Kathy Yelick, University of California, Berkeley

Presentations

- “Status Update” — Vivek Sarkar
- “Presentation” — Kathy Yelick
- “FAST-OS” — Ron Brightwell
- “Cyclops System Software” — G. R. Gao

Meeting # 6

November 11, 2008, Stanford, CA
Host: Stanford University

Committee members present

Dan Campbell, Bill Carlson (via teleconference), Andrew Chien, Bill Dally, Mootaz Elnohazy, Mary Hall, Bill Harrod, Jon Hiller, Sherman Karp, David Koester, Peter Kogge, Mark Richards, Vivek Sarkar, John Shalf, Allan Snively, Tom Sterling

Visitors

- Anant Agarwal, Massachusetts Institute of Technology

Presentations

- “Status Update” — Vivek Sarkar
- “Self-Aware Computing Presentation” — Anant Agarwal

Meeting # 7

February 25, 2009, Stanford, CA
Host: Stanford University

Committee members present

Bill Dally, Mootaz Elnohazy, Mary Hall, Bill Harrod, Jon Hiller, Sherman Karp, Peter Kogge, Mark Richards, Vivek Sarkar, John Shalf, Allan Snively, Tom Sterling

Visitors

- None

September 14, 2009

Presentations

- “Status Update” — Vivek Sarkar

Bibliography

- [1] <http://top500.org/>.
- [2] <http://www.er.doe.gov/ASCR/ProgramDocuments/TownHall.pdf>.
- [3] <http://hpcrd.lbl.gov/E3SGS/main.html>.
- [4] http://computing.ornl.gov/workshops/town_hall/.
- [5] <https://www.cels.anl.gov/events/workshops/townhall07/index.php>.
- [6] <http://www.zettaflops.org/fec07/index.html>.
- [7] <http://www.lanl.gov/roadrunner/>.
- [8] <http://www.ncsa.uiuc.edu/BlueWaters/>.
- [9] http://portal.acm.org/ft_gateway.cfm?id=1188502&type=pdf&coll=ACM&dl=GUIDE&CFID=1529677&CFTOKEN=97129850.
- [10] <http://www.gaussian.com/>.
- [11] <http://www.msg.ameslab.gov/GAMESS/>.
- [12] <http://www.simulia.com/>.
- [13] <http://www.emsl.pnl.gov/docs/global/ga.html>.
- [14] <http://www.csm.ornl.gov/workshops/HPA/documents/1-arch/hpa-xmt.pdf>.
- [15] <http://en.wikipedia.org/wiki/Little'sLaw>.
- [16] <http://www.vni.com/products/imsl/fortran/overview.php>.
- [17] http://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms.
- [18] <http://www.netlib.org/blas/>.
- [19] <http://www.netlib.org/lapack/>.
- [20] <http://math-atlas.sourceforge.net/>.
- [21] <http://icl.cs.utk.edu/plasma/index.html>.
- [22] <http://www.fftw.org/>.
- [23] <http://www.ffte.jp/>.

- [24] <http://www.vsipl.org/>.
- [25] <http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm>.
- [26] <http://www-03.ibm.com/systems/p/software/essl/index.html>.
- [27] <http://www.mpi-forum.org/docs/>.
- [28] <http://www.emsl.pnl.gov/docs/parsoft/tcgmsg/tcgmsg.html>.
- [29] <http://www.emsl.pnl.gov/docs/parsoft/tcgmsg-mpi/>.
- [30] http://www.mhpcc.edu/training/workshop2/mpi_io/MAIN.html.
- [31] <http://www.emsl.pnl.gov/docs/parsoft/ma/MA.html>.
- [32] <http://www.emsl.pnl.gov/docs/parsoft/armci/>.
- [33] <http://www.emsl.pnl.gov/docs/parsoft/chemio/chemio.html>.
- [34] <http://www.windriver.com>.
- [35] <http://www-unix.mcs.anl.gov/zeptoos>.
- [36] Anant Agarwal and Bill Harrod. Organic computing, August 2006.
- [37] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.
- [38] E. Allan et al. The Fortress language specification version 0.618. Technical report, Sun Microsystems, April 2005.
- [39] Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, Amos Waterland, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, and Livio Soares. Experience distributing objects in an smmp os. *ACM Trans. Comput. Syst.*, 25(3):6, 2007.
- [40] David A. Bader. *Petascale Computing: Algorithms and Applications*. Chapman & Hall/CRC, 2007.
- [41] Pavan Balaji, Wu-chun Feng, Jeremy Archuleta, Heshan Lin, Rajkumar Kettimuthu, Rajeev Thakur, and Xiaosong Ma. Semantics-based distributed i/o for mpiblast. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 293–294, New York, NY, USA, 2008. ACM.
- [42] Guy Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-92-103, Carnegie Mellon University, January 1992.
- [43] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 1995. ACM.
- [44] Fred Brauer and Carlos Castillo-Chavez. *Mathematical Models in Population Biology and Epidemiology*, volume 40 of *Texts in Applied Mathematics*. Springer, 2001.

- [45] Ian Buck. Brook Specification v0.2.
- [46] Zoran Budimlić, Aparna Chandramowlishwaran, Kathleen Knobe, Geoff Lowney, Vivek Sarkar, and Leo Treggiari. Multi-core implementations of the concurrent collections programming model. In *CPC '09: 14th International Workshop on Compilers for Parallel Computers*. Springer, January 2009.
- [47] Zoran Budimlic, Aparna M. Chandramowlishwaran, Kathleen Knobe, Geoff N. Lowney, Vivek Sarkar, and Leo Treggiari. Declarative aspects of memory management in the concurrent collections parallel programming model. In *DAMP '09: Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pages 47–58, New York, NY, USA, 2008. ACM.
- [48] Mary Carrington and Stephen J. O'Brien. The influence of hla genotype on aids. *Annual Review of Medicine*, 54(1):535–551, 2003. PMID: 12525683.
- [49] Mark J. Chaisson and Pavel A. Pevzner. Short read fragment assembly of bacterial genomes. *Genome Research*, 18(2):324–330, 2008.
- [50] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.
- [51] Francis S. Collins, Michael Morgan, and Aristides Patrinos. The human genome project: Lessons from large-scale biology. *Science*, 300(5617):286–290, 2003.
- [52] Willem de Bruijn and Herbert Bos. Pipesfs: fast linux i/o in the unix tradition. *SIGOPS Oper. Syst. Rev.*, 42(5):55–63, 2008.
- [53] Steven J. Deitz, David Callahan, Bradford L. Chamberlain, and Lawrence Snyder. Global-view abstractions for user-defined reductions and scans. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 40–47, New York, NY, USA, 2006. ACM.
- [54] Jack B. Dennis. Data Flow Supercomputers. *IEEE Computer*, 13(11):48–56, November 1980.
- [55] Jack Dongarra. Manycore computing: The impact on numerical software for linear algebra libraries. <http://www.netlib.org/utk/people/JackDongarra/SLIDES/sc07-wksh-mcore-1107.pdf>.
- [56] Jack Dongarra, Pete Beckman, Patrick Aerts, Frank Cappello, Thomas Lippert, Satoshi Matsuoka, Paul Messina, Terry Moore, Rick Stevens, Anne Trefethen, and Mateo Valero. The international exascale software project: A call to cooperative action by the global high performance community. *The International Journal of High Performance Computing Application*, 23(4), November 2009.
- [57] H. Carter Edwards. Software environment for developing complex multiphysics applications. 2002.
- [58] D. J. Eisenstein and P. Hut. Hop: A new group-finding algorithm for n-body simulations. *Astrophysical Journal*, 498:137–142, May 1998.

- [59] D. J. Eisenstein and P. Hut. Hop: A new group-finding algorithm for n-body simulations. *Astrophysical Journal*, 498:137–142, May 1998.
- [60] Tarek El-Ghazawi, William W. Carlson, and Jesse M. Draper. UPC Language Specification v1.1.1, October 2003.
- [61] D. R. Engler, M. F. Kaashoek, and J. O’Toole, Jr. Exokernel: an operating system architecture for application-level resource management. pages 251–266, 1995.
- [62] Peter Kogge et al. Exascale computing study: Technology challenges in achieving exascale system.
- [63] J. Fass. Personal communication, 2008.
- [64] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *SC ’06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83, New York, NY, USA, 2006. ACM.
- [65] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *PLDI ’98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM.
- [66] A. Funk, V. Basili, L. Hochstein, and J. Kepner. Analysis of parallel software development using the relative development time productivity metric. *CTWatch Quarterly*, 2(4).
- [67] Anwar Ghuloum. Ct: channelling nesl and sisal in c++. In *CUFP ’07: Proceedings of the 4th ACM SIGPLAN workshop on Commercial users of functional programming*, pages 1–3, New York, NY, USA, 2007. ACM.
- [68] G. Gilder. *Telecosm: The World After Bandwidth Abundance*. Free Press, New York, NY, 2002.
- [69] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 291–303, New York, NY, USA, 2002. ACM.
- [70] *GPFS V3.2.1 Concepts, Planning, and Installation Guide*. IBM Corporation., August 2008.
- [71] Jim Gray and Alexander S. Szalay. Where the rubber meets the sky: Bridging the gap between databases and science, 2004.
- [72] B. Graybill, G. Allen, K. Alvin, A. Atashi, M. Drazhal, D. Fisher, M. Giles, B. Lucas, T. Mattson, H. Morgan, E. Schnetter, B. Schott, E. Seidel, J. Shalf, S. Shamsian, and S.S. Tong. Hpc application software consortium summit (hpcasc) - concept paper. http://www.cct.lsu.edu/gallen/Reports/HPCASC_March2007.pdf, March 25-26, 2008.
- [73] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, May 2009.

September 14, 2009

Page 147

ECSS Report

- [74] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, 1988.
- [75] Mary Hall, David Padua, and Keshav Pingali. Compiler research: the next 50 years. *Commun. ACM*, 52(2):60–67, 2009.
- [76] Robert Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions of Programming Languages and Systems*, 7(4):501–538, October 1985.
- [77] *HDF5 User's Guide, HDF5 Release 1.8.2*. HDF Group., November 2008.
- [78] Patrick Heimbach, Chris Hill, and Ralf Giering. An efficient exact adjoint of the parallel mit general circulation model, generated via automatic differentiation. *Future Generation Computer Systems*, 21(8):1356–1371, 2005.
- [79] David Hernandez, Patrice Franois, Laurent Farinelli, Magne Ø sterås, and Jacques Schrenzel. De novo bacterial genome sequencing: Millions of very short reads assembled on a desktop computer. *Genome Research*, 18(5):802–809, 2008.
- [80] Dean Hildebrand and Peter Honeyman. Exporting storage systems in a scalable manner with pnfs. In *MSST '05: Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 18–27, Washington, DC, USA, 2005. IEEE Computer Society.
- [81] Mark D. Hill. What is scalability? *SIGARCH Comput. Archit. News*, 18(4):18–21, 1990.
- [82] Kenneth Iverson. *A Programming Language*. John Wiley and Sons, New York, NY, 1962.
- [83] Efrat Jaeger-Frank, Christopher Crosby, Ashraf Memon, Viswanath Nandigam, J. Arrow-smith, Jeffery Conner, Ilkay Altintas, and Chaitan Baru. chapter A Three Tier Architecture for LiDAR Interpolation and Analysis, pages 920–927. 2006.
- [84] L Kale and S Krishnan. CHARM++: A Portable Concurrent Object-Oriented System based on C++. *ACM Sigplan Notices: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 28(10):91–108, October 1993.
- [85] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
- [86] Michael Kistler, John Gunnels, Daniel Brokenshire, and Brad Benton. Petascale computing with accelerators. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 241–250, New York, NY, USA, 2009. ACM.
- [87] David Koester. Exascale study application footprints. https://pastec.gtri.gatech.edu/ECSS/images/c/c5/Exascale_App_Footprints_07-1449.pdf, October 2007.
- [88] Mike Kravetz and Hubertus Franke. Implementation of a multi-queue scheduler for linux. <http://lse.sourceforge.net/scheduling/mq1.html>, 2001.
- [89] Andres Kriete and Roland Eils. *Computational Systems Biology*. Academic Press, 2005.
- [90] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. *Conference Record of 8th ACM Symposium on Principles of Programming Languages*, 1981.

- [91] S. Kumar, H. Raj, K. Schwan, and I. Ganey. The sidecore approach to efficient virtualization in multicore systems. 2007.
- [92] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.
- [93] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
- [94] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, October 1998.
- [95] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–11, New York, NY, USA, 2007. ACM.
- [96] W. B. Ligon and R. B. Ross. Implementation and performance of a parallel file system for high performance distributed applications. In *HPDC '96: Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, page 471, Washington, DC, USA, 1996. IEEE Computer Society.
- [97] Ying Liu, Wei keng Liao, and Alok Choudhary. Design and evaluation of a parallel hop clustering algorithm for cosmological simulation. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 82.1. IEEE Computer Society, 2003.
- [98] William Loring, Lee Harland, and Bryn Williams-Jones. High-throughput electronic biology: mining information for drug discovery. *Nature Reviews Drug Discovery*, 6(3):220–230.
- [99] *Lustre 1.6 Operations Manual*. Sun Microsystems., Nov 21 2008.
- [100] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. SISAL: Streams and Iteration in a Single Assignment Language Reference Manual Version 1.2. Technical report, Lawrence Livermore National Laboratory, 1985. No. M-146, Rev. 1.
- [101] Paul E. McKenney and Jonathan Walpole. Introducing technology into the linux kernel: a case study. *SIGOPS Oper. Syst. Rev.*, 42(5):4–17, 2008.
- [102] M Metcalf and J Reid. *Fortran 90 Explained*. Oxford Science Publications, Oxford, England, 1990.
- [103] R. G. Minnich, M. J. Sottile, S.-E. Choi, E. Hendriks, and J. McKie. Right-weight kernels: an off-the-shelf alternative to custom light-weight kernels. *SIGOPS Operating Systems Review*, 40, 2006.
- [104] E Mohr, D Kranz, and JR. Halstead, R. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

- [105] José Moreira, Michael Brutman, José Castanos, Thomas Engelsiepen, Mark Giampapa, Tom Gooding, Roger Haskin, Todd Inglett, Derek Lieber, Pat McCarthy, Mike Mundy, Jeff Parker, and Brian Wallenfelt. Designing a highly-scalable operating system: the blue gene/l story. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 118, New York, NY, USA, 2006. ACM.
- [106] *MPI: A Message-Passing Interface Standard, Version 2.1*. Message Passing Interface Forum., June 2008.
- [107] David Nagle, Denis Serenyi, and Abbie Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 53, Washington, DC, USA, 2004. IEEE Computer Society.
- [108] Jarek Nieplocha, Holger Dachsel, and Ian Foster. Distant i/o: One-sided access to secondary storage on remote processors. In *HPDC '98: Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, page 148, Washington, DC, USA, 1998. IEEE Computer Society.
- [109] Robert W. Numrich and John Reid. Co-Array Fortran for parallel programming. *ACM SIGPLAN Fortran Forum Archive*, 17:1–31, August 1998.
- [110] J. Michael Oakes and Jay S. Kaufman. *Methods in Social Epidemiology: Research Design and Methods*. Wiley Default, 2006.
- [111] R.A. Oldfield, P. Widener, A.B. Maccabe, L. Ward, and T. Kordenbrock. Efficient data-movement for lightweight i/o. *Cluster Computing, IEEE International Conference on*, 0:1–9, 2006.
- [112] Swapnil V. Patil, Garth A. Gibson, Sam Lang, and Milo Polte. Giga+: scalable directories for shared file systems. In *PDSW '07: Proceedings of the 2nd international workshop on Petascale data storage*, pages 26–29, New York, NY, USA, 2007. ACM.
- [113] F. Petrini, D. Kerbyson, and S. Pakin. The case of missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ascii q. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC03)*, 2003.
- [114] Removing the big kernel lock. http://kerneltrap.org/Linux/Removing_the_Big_Kernel_Lock, May 2008. Viewed on September 27, 2008.
- [115] Russ Rew, Glenn Davis, Steve Emmerson, Harvey Davies, and Ed Hartnett. *The NetCDF Users Guide. NetCDF Version 4.0.1*. Unidata Program Center., March 2009.
- [116] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.
- [117] Vivek Sarkar. Automatic Partitioning of a Program Dependence Graph into Parallel Tasks. *IBM Journal of Research and Development*, 35(5/6), 1991.
- [118] Vivek Sarkar. The PTRAN Parallel Programming System. In B. Szymanski, editor, *Parallel Functional Programming Languages and Compilers*, ACM Press Frontier Series, pages 309–391. ACM Press, New York, 1991.

- [119] Vivek Sarkar. Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers. *IBM Journal of Research and Development*, 41(3), May 1997.
- [120] Vivek Sarkar, William Harrod, and Allan E. Snively. Software challenges in extreme scale systems. In *2009 Conference on Scientific Discovery through Advanced Computing Program (SciDAC)*, June 2009.
- [121] E. Schnetter, C. D. Ott, G. Allen, P. Diener, T. Goodale, T. Radke, E. Seidel, and J. Shalf. Cactus framework: Black holes to gamma ray bursts. 2007.
- [122] Charles Semple and M. A. Steel. *Phylogenetics*. Oxford University Press, 2003.
- [123] John Shalf, Shoaib Kamil, Leonid Oliker, and David Skinner. Analyzing ultra-scale application communication requirements for a reconfigurable hybrid interconnect. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 17, Washington, DC, USA, 2005. IEEE Computer Society.
- [124] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer III. Phaser accumulators: a new reduction construct for dynamic parallelism. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, May 2009.
- [125] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2008. ACM.
- [126] Jun Shirako, Jisheng Zhao, V. Krishna Nandivada, and Vivek Sarkar. Chunking parallel loops in the presence of synchronization. In *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*. ACM, June 2009.
- [127] Stephen C. Simms, Gregory G. Pike, S. Teige, Bret Hammond, Yu Ma, Larry L. Simms, C. Westneat, and Douglas A. Balog. Empowering distributed workflow with the data capacitor: maximizing lustre performance across the wide area network. In *SOCF '07: Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches*, pages 53–58, New York, NY, USA, 2007. ACM.
- [128] S. Skedzielewski and J. Glauert. IF1 – An Intermediate Form for Applicative Languages. Technical report, Lawrence Livermore National Laboratory, 1985. No. M-170.
- [129] A. Snively, R. Pennington, and R. Loft. Workshop report: Petascale computing in the biological sciences. www.sdsc.edu/~allans.
- [130] A. Snively, R. Pennington, and R. Loft. Workshop report: Petascale computing in the geosciences. www.sdsc.edu/~allans.
- [131] MSC Software. Simenterprise extending simulation to the enterprise. 2007.
- [132] Guy Steele. The Future Is Parallel: What's a Programmer to Do? Breaking Sequential Habits of Thought. One-hour talk presented at the 5 March 2009 meeting of the New England Programming Languages and Systems (NEPLS) Symposium at Mitre. <http://research.sun.com/projects/plrg/Publications/NEPLSMarch2009Steele.pdf>, 2009.

- [133] *Draft OSD Standard. T10 Committee.* Storage Networking Industry Association (SNIA)., July 2004.
- [134] Alexander Szalay and Jim Gray. 2020 computing: Science in an exponential world. *Nature*, 440(7083):413–414, Mar 2006.
- [135] Nathan R. Tallent and John M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 229–240, New York, NY, USA, 2009. ACM.
- [136] Nathan R. Tallent, John M. Mellor-Crummey, and Michael W. Fagan. Binary analysis for measurement and attribution of program performance. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 441–452, New York, NY, USA, 2009. ACM.
- [137] Ani R. Thakar. Lessons learned from the sdss catalog archive server. *Computing in Science and Engineering*, 10(6):65–71, 2008.
- [138] Christophe Tretz. Cmos transistor sizing for minimization of energy-delay product. In *GLSVLSI '96: Proceedings of the 6th Great Lakes Symposium on VLSI*, page 168, Washington, DC, USA, 1996. IEEE Computer Society.
- [139] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
- [140] Bryan Veal and Annie Foong. Performance scalability of a multi-core web server. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 57–66, New York, NY, USA, 2007. ACM.
- [141] Anh Vo, Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur. Formal verification of practical mpi programs. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 261–270, New York, NY, USA, 2009. ACM.
- [142] Thorsten von Eicken, David E. Culler, Klaus Erik Schauser, and Seth Copen Goldstein. Retrospective: active messages: a mechanism for integrating computation and communication. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 83–84, New York, NY, USA, 1998. ACM.
- [143] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Miller, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.
- [144] Michael E. Wolf and Monica S. Lam. A Data Locality Optimization Algorithm. *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 30–44, June 1991.
- [145] Michael E. Wolf and Monica S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

- [146] C. Wunsch and P. Heimbach. Practical global oceanic state estimation. *Physica D Nonlinear Phenomena*, 230:197–208, June 2007.
- [147] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing, LCPC'09*, 2009.
- [148] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and performance using partitioned global address space languages. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32, New York, NY, USA, 2007. ACM.
- [149] D.G. York. The sloan digital sky survey astronomical journal. <http://www.sdss.org/>, 2000.
- [150] Daniel R. Zerbino and Ewan Birney. Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 18(5):821–829, 2008.
- [151] Shujia Zhou, Amidu Oloso, Megan Damon, and Tom Clune. Application controlled parallel asynchronous io. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 178, New York, NY, USA, 2006. ACM.

APPENDIX C

FINAL REPORT OF EXASCALE COMPUTING RESILIENCY STUDY

Technical Report, Contract FA8650-07-C-7724
ECE Project E-21-67V (PeopleSoft ID 210667)



**Exascale Computing Study Report:
Software Resiliency Study**
Reporting Period: April 2008 – October 2008

By:

Mark A. Richards
School of Electrical and Computer Engineering



Submitted to:
AFRL/SNDI
Building 620, Room 3 DU57
2241 Avionics Circle
Wright-Patterson AFB, OH 45433-7320
ATTN: Ms. Kerry Hill

Submitted by:
GEORGIA INSTITUTE OF TECHNOLOGY
A Unit of the University System of Georgia
Georgia Tech Research Institute
Atlanta, Georgia 30332-0800

Cage No. 1G474

Contracting through:
GEORGIA TECH RESEARCH CORPORATION
Centennial Research Building
Georgia Institute of Technology
Atlanta, Georgia 30332

November 2008

**The views expressed are those of the authors and do not reflect the
official policy or position of the Department of Defense or the U.S. Government.**

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ECRS Report

TABLE OF CONTENTS

<u>Paragraph</u>	<u>Page</u>
PREFACE	ii
SECTION 1: EXASCALE COMPUTING STUDIES	1
1.1 Introduction	1
1.1.1 ECSS	1
1.1.2 ECRS	2
1.2 Summary of ECRS Study Focus and Results	2
SECTION 2: RESILIENCY STUDY	3

PREFACE

This document is a release of the Exascale Computing Study technical report documenting the Software Resiliency Study. This report was prepared under Georgia Institute of Technology (GIT) Project E-21-67V, "Exascale Computing Study." This project is sponsored by the Defense Advanced Research Projects Agency (DARPA), contracting through the Air Force Research Laboratory (AFRL), and is conducted under AFRL contract FA8650-07-C-7724.

The author would like to thank DARPA, the U. S. Air Force, Dr. William Harrod of DARPA, and Ms. Kerry Hill of AFRL for their support of this work.

SECTION 1: EXASCALE COMPUTING STUDIES

1.1 Introduction

An ExaScale computer system is a computer system with approximately a one thousand-fold increase in capabilities relative to the computer systems currently under development by DARPA's High Productivity Computing Systems (HPCS) program. In 2007-2008, DARPA has conducted an Exascale Computing Study addressing hardware and system architecture issues in the development of exascale computing systems. Georgia Tech (GT) has provided technical support to that study under this contract.

Two critical issues identified by the ECS are the development of effective parallel programming methodologies for systems having extreme degrees of concurrency, and exascale system resiliency. In 2008, the original ECS study was augmented with two supplemental studies to further define the technologies and investments needed by 2010 to enable the development of ExaScale computing systems by 2015. These two additional studies are denoted the ExaScale Computing Software Study (ECSS) and the ExaScale Computing Resiliency Study (ECRS).

1.1.1 ECSS

The ECSS, which is still ongoing at this writing, convened a group of academic and industry experts to explore the issue of effective programming for extreme concurrency by addressing at least the following issues:

- Intelligent management of the increasingly complex computational resources
- Energy requirements for the execution of an application code
- Multiple classes of computing systems: ExaScale data centers, PetaScale departmental systems, and TeraScale embedded systems

Key study topics include:

- Development Environment: Languages, compilers, tools, runtime systems
- Applications: Challenges, characteristics, & benchmarks
- Energy: Managing data movement to optimize energy and parallel efficiency
- System architectures: Concurrency, memory hierarchies, protocols, functional partitioning, execution models

The goal of the ECSS study is the development of a technical report that

- Identifies the key software technologies, approaches, and methodologies that must be in place to effectively utilize the extreme levels of concurrency expected in Exascale computing technology, and
- Specifies the research problems that must be solved so that these key technologies can be in place in time to support initial deployment of Exascale technology in 2015.

1.1.2 ECRS

The ECRS, which has been completed, convened a group of academic and industry experts to explore the issue of resiliency of ExaScale systems with the principal goal of achieving a 1000x improvement in mean time to failure (MTTF) scalability for ExaScale systems by avoiding global checkpointing as used in present-day systems. Key study topics included:

- Programming using transactional models and network-accessible memories
- Effective use of CMOS stable semiconductor memories and optical links

The goal of the ECRS study was the development of an extended technical white paper that

- Identifies the key technologies, approaches, and methodologies that must be in place to ensure scalability of MTTF in ExaScale systems, and
- Specifies the research problems that must be solved so that these key technologies can be in place in time to support initial deployment of Exascale technology in 2015.

The Editor and Study Lead of the ECRS was Dr. E.N. (Mootaz) Elnozahy, IBM, Austin. The ECRS study committee consisted of the following subject matter experts:

Prof. Tarek El-Ghazawi	George Washington University
Prof. Armando Fox	University of California
Forest Godfrey	Cray
Dr. Adolfo Hoisie	Los Alamos National Laboratory
Prof. Kathryn McKinley	University of Texas
Prof. Rami Melhem	University of Pittsburgh
Prof. James Plank	University of Tennessee
Dr. Partha Ranganathan	HP Labs
Josh Simons	Sun Microsystems

This report documents the results of the ECRS study.

1.2 Summary of ECRS Study Focus and Results

Today, 20% or more of the computing capacity in a large high-performance computing system is wasted due to failures and recoveries. Typical MTBF is from 8 hours to 15 days. As systems increase in size to field petascale computing capability and beyond, the MTBF will go lower and more capacity will be lost.

The ECRS study analyzed the current problem in reliable systems and suggests new avenues for research in resilient systems at extreme scale. It showed that central to the current problem in providing reliability is the programming model based on flat message passing using MPI. This model does not offer any failure containment, and thus a failure in one node in the system triggers an entire system failure. As systems continue to increase in size, this is not tenable.

The study also analyzed the new trends in technology and showed that power management, the recent trend toward heterogeneous computing and the expected increase in system size all will interact negatively with resilience at the system level.

The study proposed new avenues in research that have a promise of mitigating the situation. This approach consists of two parts, one is to exploit existing technology trends such as the proliferation of multi-core systems and flash memory, and the expanded use of virtualization. The second part consists of opening new areas of research based on exploiting new emerging programming models, as well as system-level implementation of resilience. New directions in statistical machine learning and compiler and run-time support for resilience are also suggested.

The criterion of success in the proposed research is to reduce the computing capacity that is wasted due to failure from today's 20% to within 2% in future systems of larger size.

SECTION 2: RESILIENCY STUDY

The detailed analysis, conclusions, and recommendations of the ECRS study are given by the System Resilience at Extreme Scale White Paper, attached on the following pages. This white paper was delivered to DARPA by Dr. Elnohazy on or about November 9, 2008.

System Resilience at Extreme Scale

White Paper

Contributors:

Professor Ricardo Bianchini, Rutgers University, Piscataway
Professor Tarek El-Ghazawi, George Washington University, Washington D.C.
Professor Armando Fox, University of California, Berkeley
Forest Godfrey, Cray, Minneapolis
Dr. Adolfo Hoisie, Los Alamos National Laboratory, Los Alamos
Professor Kathryn McKinley, University of Texas, Austin
Professor Rami Melhem, University of Pittsburgh, Pittsburgh
Professor James Plank, University of Tennessee, Knoxville
Dr. Partha Ranganathan, HP Labs, Palo Alto
Josh Simons, Sun Microsystems, Cambridge

Editor and Study Lead:

Dr. E.N. (Mootaz) Elnozahy, IBM, Austin

**Prepared for Dr. William Harrod, Defense Advanced Research Project
Agency (DARPA).**

Executive Summary

Today, 20% or more of the computing capacity in a large high-performance computing system is wasted due to failures and recoveries. Typical MTBF is from 8 hours to 15 days. As systems increase in size to field petascale computing capability and beyond, the MTBF will go lower and more capacity will be lost.

This document analyzes the current problem in reliable systems and suggests new avenues for research in resilient systems at extreme scale. We show that central to the current problem in providing reliability is the programming model based on flat message passing using MPI. This model does not offer any failure containment, and thus a failure in one node in the system triggers an entire system failure. As systems continue to increase in size, this is not tenable.

We also analyze the new trends in technology and show that power management, the recent trend toward heterogeneous computing and the expected increase in system size all will interact negatively with resilience at the system level.

We then propose new avenues in research that have a promise of mitigating the situation. This approach is depicted in the figure below. It consists of two parts, one is to exploit existing technology trends such as the proliferation of multi-core systems and flash memory, and the expanded use of virtualization. The second part consists of opening new areas of research based on exploiting new emerging programming models, as well as system-level implementation of resilience. New directions in statistical machine learning and compiler and run-time support for resilience are also suggested.

The criterion of success in the proposed research is to reduce the computing capacity that is wasted due to failure from today's 20% to within 2% in future systems of larger size.

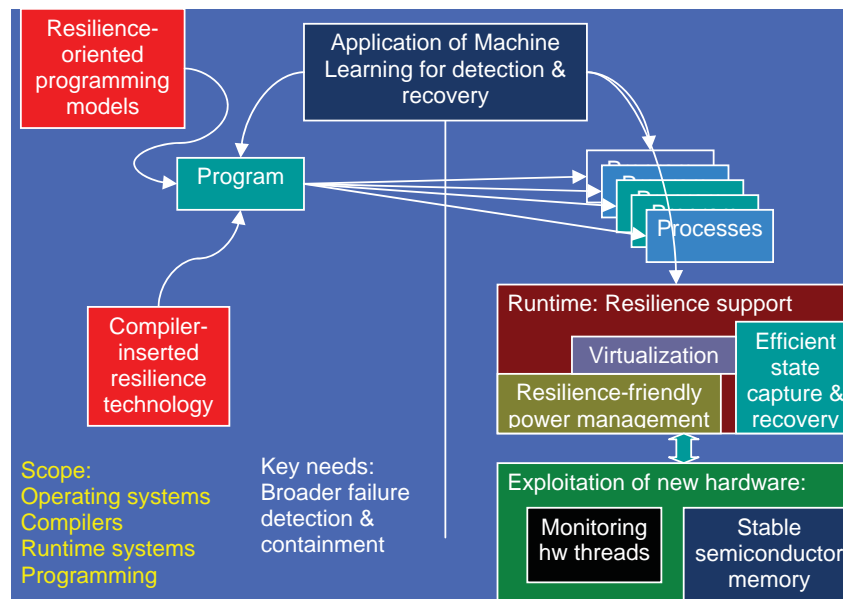


Table of Contents

1	Introduction.....	4
2	Understanding Failures	5
2.1	Failures in Commercial Systems	6
2.2	Failures Sources in Large-Scale Systems	7
2.3	Failures and Scalability.....	8
2.4	Current State of the Art: Checkpoint and Rollback.....	9
2.5	Checkpointing Implementations	11
2.5.1	Checkpointing Frequency	11
2.5.2	Possible Optimizations.....	12
2.5.3	Current State of the Art.....	13
3	Future Trends and Impact on Resilience	13
3.1	Power Management	14
3.2	System Size	16
3.3	Heterogeneity	16
4	Plausible Fields of Research	17
4.1	Exploiting New Technology Trends	17
4.1.1	Exploiting Multi-Core Systems	17
4.1.2	Exploiting Semiconductor-based Stable Storage.....	20
4.1.3	Exploiting Virtualization	21
4.2	New Technologies	21
4.2.1	New Programming Models.....	21
4.3	Application of Machine Learning.....	22
4.4	Compiler Support for Resilience	23
5	Bibliography	23

1 Introduction

As systems continue to grow in size and capabilities, the current state of the art in system reliability is being pushed to its limit. This document summarizes the current and future trends in large-scale systems, with emphasis on how they affect the reliability problem. It then presents potential venues for research and innovations that exploit these trends and improve the reliability of the system in the face of current and anticipated future problems.

At the dawn of the Peta-scale computing era, it is remarkable that the current state of the art in reliability for large-scale systems has barely advanced beyond the concepts of the early 1990's. The current prevailing programming model in large-scale systems is based on message passing, with coordinated periodic checkpointing the method of choice to provide fault tolerance in practical installations. Coordinated checkpointing worked well for early ASC machines such as ASC Blue and ASC White. But the performance and recoverability of these techniques are inadequate for modern systems that include 100,000's of cores. Notable are the performance overhead due to saving the entire system state periodically to a centralized stable storage, and the failure of these methods to protect healthy nodes from the effects of failed nodes, leading to cascading aborts throughout the system due to a single node failure. The performance overhead is due to the need to save the state of the application to stable storage made out of a centralized, slow and expensive disk storage facility. This state size is measured in Tera Bytes and soon in Peta Bytes, while disk speeds have barely improved in the past decade. A member of the team has related that current systems in his organization are typically stopped for about 10 minutes each hour to perform the checkpointing operation. This translates to about 16% performance and power waste in failure-free operation alone. In the Peta-scale era, the increase in size and the reduction of the Mean Time Between Failure (MTBF) of future systems will only push the frequency of checkpointing higher, in turn increasing the overhead in power and performance and exacerbating the pressure on the network and the stable storage devices. These projections are not tenable.

Reliability in future systems will also be affected by the system interaction with power consumption. At the current rate of 3MW/PF, multi-petaflop systems will require data centers that are capable to provide 10's of megawatts from the power grid. System vendors will have to deploy power management technologies to minimize the energy spent in a given computation, but the impact of these technologies on system reliability is not at all understood. For example, exploiting idle cycles in the system to reduce power will reduce system temperature, which in turn reduces leakage and improves immunity against soft errors in the electronic components. However, the continuous change in voltage and frequency will introduce thermal and concomitant mechanical stresses on the electronic chips and board-level electrical connections. These stresses may cause device damage and failures. Another example is power management techniques for disks. Today's enterprise-level disks that are typically used in large-scale systems are not designed to withstand many power cycles. Any power management technique therefore is bound to reduce the Mean Time to Failure of the storage subsystem. If one considers that many applications in a large-scale system have synchronized computation, file access and communication phases. The effect of employing power management in for these applications will subject the power grid in the data center to power swings measured in megawatts over as little as microseconds. It is not even clear that the current design of data centers can sustained such swings.

We also observe a trend toward integrating the data center facilities into the computing systems. For example, future large scale systems will likely use water cooling, with the water being drawn from the data center cooling subsystem. The computer management subsystem will expand beyond the traditional boundaries of the computing system to include control over the data center cooling and thermal management. Thus, new modes of failures are likely to be present as the failure in the devices that were traditionally outside the domain of the computing system will be manifested directly into system outages. Furthermore components such as board-level fans and power supplies will be stressed by the variation due to power management, and these may accelerate their failure rates. Current reliability models do not address these problems, and one must conclude then that the interactions between reliability and power is not understood, and the magnitude of the problem must be studied.

Another trend that we predict is the heterogeneity of future systems. The only Peta-scale system that exists today is composed of two heterogeneous processors. Other accelerators such as Field Programmable Arrays (FPGA), vector accelerators and special-purpose computers will be common in future systems. These accelerators may not run a conventional operating system, may not offer the usual detection of failures that we are used to in conventional architectures, and it may not be straightforward to capture their state into the system's state as part of a checkpoint, for instance. Handling the failure itself is not well understood. For example, if a failure in an FPGA occurs, do we declare the entire machine as failed, even though the "main node" remains healthy?

Even the question of what constitutes a failure will be an important field of study. Today, the reliability technology that we have handles only what is commonly referred to as "crash" failures. In reality, system crashes result from detected hardware failures that force a system check, or from operating system failures (e.g. hangs). But the systems of the future will be complex, where a computation unit or a node will consist of possibly heterogeneous multicore chips, and it is worth exploring if a partial failure in the node could be handled in a manner that would allow the node to continue operation, albeit at a reduced performance while recovery is attempted. In other words, a "crash" may not be the only type of failure that the system should detect and recover from. Partial failures must be contained and tolerated, and every effort must be made to prevent the partial failure of a node from propagating throughout the system. This also should extend the ability of the system to detect and recover from the effects of soft error beyond the traditional machine check. Soft errors occur in hardware and their rate is likely to increase because of the continuous shrinking of computer chips. It is both desirable and necessary to avoid converting such errors into machine crashes as occurs today. The system should be able to detect and partially recover from such errors, so as to contain the effect of failures over the entire system. The same applies to failures that occur to the software. With the increasing complexity of having virtualized system software and potentially multiple images of operating system within a node, it will be desirable to ensure that the state of each operating system is continuously monitored and any anomalous situations detected and fixed in a manner that does not force the application to running on that node to fail.

2 Understanding Failures

Computer system failures occur due to a variety of sources, including hardware, software and human errors. Many studies have been conducted to understand the nature of these failures, their

occurrence frequency and their impact on system's mission. These failures impact the system in different ways, depending on the nature of the solution deployed and its size.

2.1 Failures in Commercial Systems

It is useful to consider how failures are handled in the commercial domain before we delve into how they are handled in large-scale systems that are typically used for scientific computing. In the commercial domain, data processing systems have a typical mission of conducting a large number of relatively independent operations that affect a large, shared database. A typical commercial system consists of a 3-tiered structure, namely the presentation or Web layer, the application or logic layer, and the data or database layer (see **Error! Reference source not found.** [1]). This popular model has become the prevailing one in commercial applications.

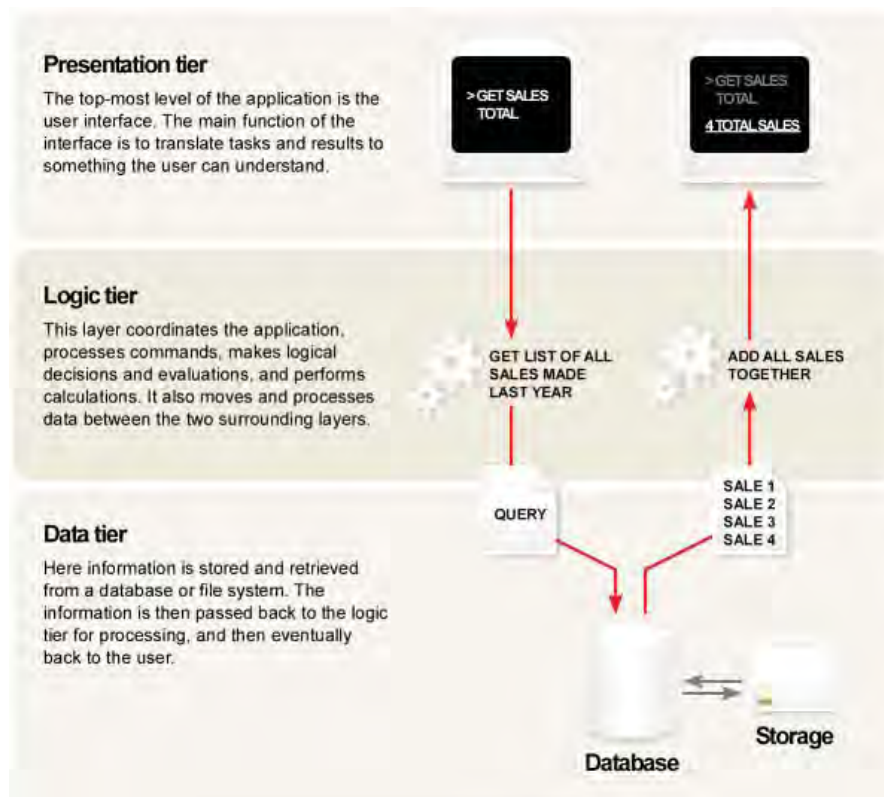


Figure 1. An example of a 3-tiered commercial application.

Hardware failures, software bugs and human errors affect commercial systems. Over the years, a certain practice has developed in which the application is structured as a collection of short running *transactions* that perform atomic updates to a replicated or data-redundant database [1]. A transaction processing system is provided to guarantee the so-called ACID properties, namely Atomicity, Consistency, Isolation and Durability. Replication and other techniques have been developed to make the database highly available in face of failures. These properties provide a

very useful abstraction to the transaction writer: A transaction takes place in an ideal failure-free environment in which the transaction appears to operate on durable data in isolation of other transactions. Failures have an all or nothing effect—either the transaction completes or it has never run, and a failed transaction never affects the database. The transaction processing system transparently manages concurrency control and failure atomicity.

Transactions enable the programmer to avoid the issues of concurrency control and failure recovery to a large degree and thus simplify application programming. And because transactions are short, their failures do not have a high cost associated with them. Of course, transaction processing impacts performance, and data redundancy consumes additional storage and data bandwidth resources. Furthermore, as more transactions attempt to manipulate shared data, the scalability of the system is tested.

Yet, despite of the costs associated with transaction processing, it has become the method of choice in commercial data processing systems. One of its particular advantages is that a failure tends to have an isolated effect on the system. A failure affects only a limited number of transactions, which can be restarted with relatively a small cost. Thus, a failure is not an impediment to scalability. In practice, we see that transactions enable commercial systems to be built at a very large scale. This is in stark contrast with scientific computing systems as will be discussed.

2.2 Failures Sources in Large-Scale Systems

Large-scale systems are relatively few and inaccessible, and therefore studies to assess their reliability and failures are rare. Recently, Schroeder and Gibson published a study on the sources of failures in two large-scale parallel systems [4]. The study surveyed over 22 production-level systems of different sizes and processor types at the Los Alamos National Laboratory. The study also included one large Non-Uniform Memory Access (NUMA) system of 512 processors per node. Various failures have been observed and tabulated. Failures included hardware failures in CPU and memory; software failures in operating systems, parallel file systems, middleware and applications; and human errors in system configuration and administration. Figure 1 shows distribution of failure sources and their respective frequencies in two different systems in the study.

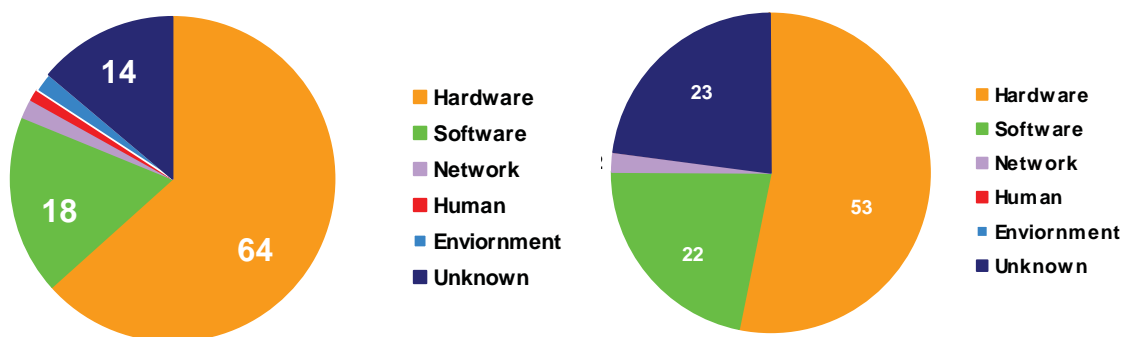


Figure 2 Failures in two sample systems.

The data show that in the two systems, hardware failures dominate the other sources. This contrasts with previous studies in commercial systems which have shown software and human errors as the dominating sources, while hardware failures tend to be limited and declining over time [1]. We believe that there are two reasons for the dominance of hardware failures in a large-scale system devoted to scientific computing. First, the software stack tends to be relatively simple. Most programs tend to be numeric intensive, rarely exercise the operating system, and the only middleware they use is the Message Passing Interface (MPI). Therefore, it is reasonable to expect that the simplicity of the software will cause relatively fewer failures, in contrast with commercial systems in which software complexity is rampant. The second reason for the dominance of hardware failures is the sheer scale of the system. With a very large number of hardware components, the system-level probability that one of them fails will be more significant than in the smaller scale commercial systems that were studied before.

The study shows also that failures occur with a surprising frequency. It appears that compute-intensive applications stress system hardware in ways that commercial systems do not. The systems under study had Mean Time Between Failures (MTBF) ranging from eight hours to 15 days, and a Mean Time To Repair (MTTR) ranging from about one hour to one day. The large MTTR time requires that large-scale systems must be provisioned with spare compute nodes that can be brought on line immediately to compensate for the nodes that fail. Otherwise, the computation will have to run on a fewer number of nodes. Often, this is simply not possible given that many applications for examples require a number of computing nodes that is a power of 2, or have built-in load management and distribution algorithms that will break if the number of machines is reduced. Generally, application code is not written to show flexibility in face of failures, and thus the failed components must be replaced for the computation to continue. This is a common practice in large installations. This provisioning, however, is a waste of resources if no failure occurs.

2.3 Failures and Scalability

A particular problem that compute-intensive parallel systems suffer from is the lack of *failure containment*. Programs in these environments tend to deploy a large number of nodes to

implement a single computation, and use MPI with a flat model of message exchange in which any node can communicate with another. As a result, a node that participates in a computation acquires dependencies on the states of the other nodes. A failure in one node, thus, is a failure of the entire computation, since the computation cannot continue until the failed node is brought back to a state that is consistent with all the nodes in the computation. But repairing the node may require other healthy nodes to roll back their state to regenerate messages that are necessary for repair. As a result, all nodes that participate in the computation may have to roll back because one of them failed. A theory of distributed system has been developed to reason about this problem [2]. This theory states that fundamentally, message-passing systems are complex because messages induce inter-process dependencies during failure-free operation. Upon a failure of one or more processes in a system, these dependencies may force some of the processes that did not fail to roll back, creating what is commonly called *rollback propagation*. To see why rollback propagation occurs, consider the situation where a sender of a message m rolls back to a state that precedes the sending of m . The receiver of m must also roll back to a state that precedes m 's receipt; otherwise, the states of the two processes would be *inconsistent* because they would show that message m was received without being sent, which is impossible in any correct failure-free execution. Under some scenarios, rollback propagation may extend back to the initial state of the computation, losing all the work performed before a failure. This situation is known as the *domino effect*.

Several solutions have been developed for the problem of recovering from a failure in a parallel systems based on message passing (such as MPI). All these solutions deploy some form of checkpointing during failure free operation, with an assortment of message logging variations that offer different tradeoffs between the performance impact of checkpointing and logging during free failure operation on one hand, and the extent of rolling back among healthy node upon a failure on the other hand. The checkpoint typically is taken to a stable storage device that is redundant and highly available. A network storage system based on disks and equipped with redundant connections to the system is typically used for the purpose of storing the checkpoints from the various nodes in the system. We note here that taking the checkpoint to a local disk within a node is not a good solution because if that node becomes disabled due to a failure, the disk is no longer accessible, and the computation cannot be restarted.

The lack of failure containment and the fact that one node failing may affect the entire computation limits the scalability of the system. If the probability of one node failing remains constant, increasing the system size simply increases the probability of a failure in one of its nodes. Ultimately, the rate of failures can be such that the computation ceases to make any progress, suffering from a repeated occurrence of failures and repairs. Given that the computations are often long-running and may exceed the time during which the system stays healthy, it follows that failures and recoveries may impede the scalability of the system. We now turn to how these issues are being solved today and the current state of the art in dealing with failures.

2.4 Current State of the Art: Checkpoint and Rollback

All available studies have shown that writing the state of a process to stable storage is the largest contributor to the performance overhead of checkpointing [2]. The simplest way to save the state of a system is to suspend execution of *all* processes at all nodes, wait for all messages that are currently in transit to reach their destinations, wait for all message queues to be emptied, then

save to stable storage the process's address space, register values, and all necessary data to reconstruct the process if necessary in the future. The execution is then resumed. This scheme can be costly for programs with large address spaces if stable storage is implemented using magnetic disks, as it is the custom.

The above mechanism is often referred to as system-level checkpointing. If a failure occurs in any node in the system, all processes are restored from the state that was last saved on stable storage. Execution then resumes from that point.

Figure 3 shows a timeline of system-level checkpointing regularly during failure-free (or normal) operation and restarting from an earlier saved state if a failure occurs.

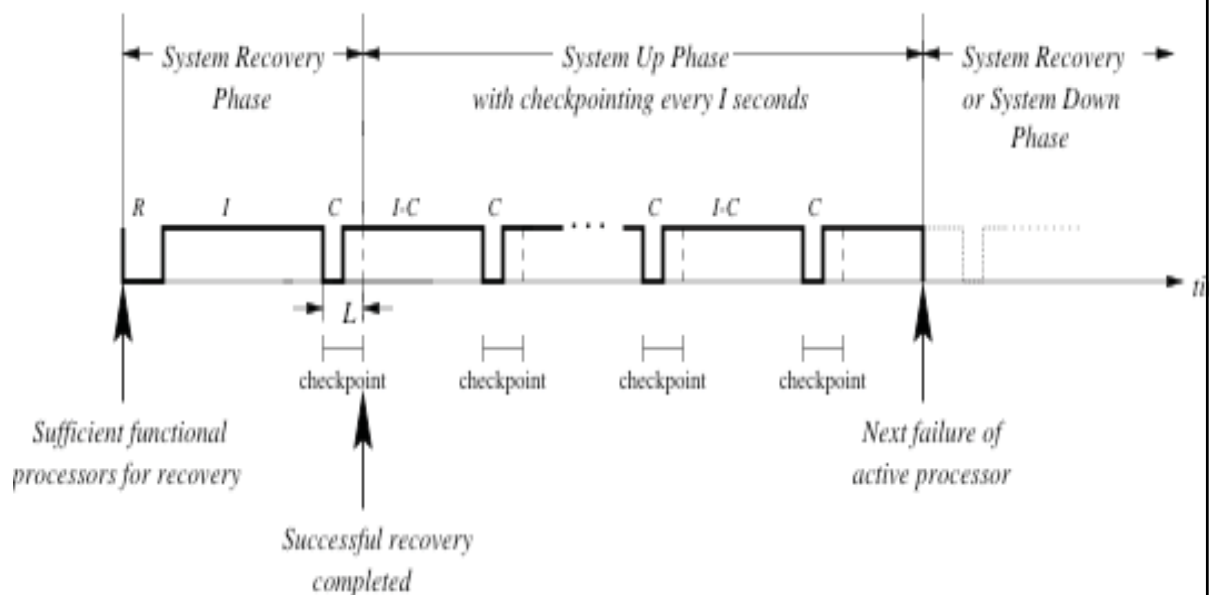


Figure 3 Failure and recovery action over time.

System-level checkpointing is expensive, and yet it is currently the state of the art in implementing checkpointing in production systems. The expense is due to stable storage access and the fact that the system stops doing useful work while the checkpoint is taken. Consider a system where the amount of main memory available on the aggregate is 1 Petabytes. It would take 1000 seconds for a sophisticated stable storage device that can store data at an effective rate of 1 Terabyte/second. This is obviously absurd, and shows that the next generation parallel systems will have to pay a tremendous premium on provisioning stable storage systems. Post petascale systems will require even more expensive resources in data bandwidths and stable storage bandwidth. Clearly, the situation is not tenable.

2.5 Checkpointing Implementations

2.5.1 Checkpointing Frequency

There is a tradeoff between the frequency of checkpointing and cost of rollback during failures. More frequent checkpoints reduce the amount of work that is at risk to be lost due to a failure. If a system takes a checkpoint every hour, then on average, about half-an-hour worth of work could be lost if a failure occurs. A large amount of work has been devoted to analyzing and deriving the optimal checkpointing frequency and placement. The problem is usually formulated as an optimization problem subject to constraints. The current practice however considers the MTBF, the amount of overhead of checkpointing and the amount of work at risk as the main driving factors in choosing the frequency of checkpointing. We explain how this is typically done by an example.

If the system is expected to have an MTBF of 1 day, then in theory the minimum number of checkpoints is twice that rate per day to ensure that the system will ultimately finish the computation. However, failures do not occur exactly according to the stated MTBF. Also, the user may not be comfortable with the notion of losing half-a-day worth of work due to a failure. Therefore, in situation of this sort, a more frequent checkpointing rate is typically desired to limit the amount of work at risk. Today, a checkpoint on the hour is the typical frequency deployed in many systems. It is also reported that for current systems of 100TF or more, a checkpoint typically requires 10 minutes to complete.

As the system scale increases, the MTBF will go down, requiring or forcing a higher frequency of checkpointing. Eventually, the system may be simply bound in taking checkpointing with very little work done if the MTBF goes down to a few hours. This situation is depicted by

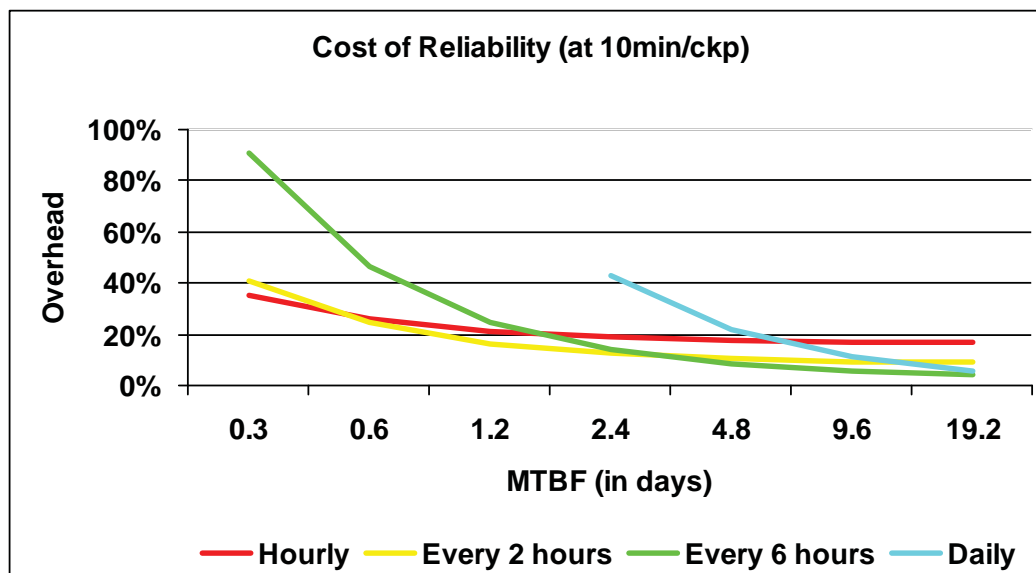


Figure 4. Cost of reliability as a function of MTBF.

Figure 3 showing how the overhead of checkpointing varies with the MTBF. The overhead in the figure takes into account the expected overhead due to failure restart. As seen from the

figure, as the MTBF drops to 8 hours, the system is expected to spend 40% of its time doing checkpointing and recovery if a checkpoint is taken on the hour. Less frequent checkpoints, say at one/6 hours, will cost the system more than 90% of its time, most likely because of the checkpointing rate leaves a lot of work at risk due to the high frequency of failures. This is not tenable, and is another aspect of how the current practice is limited in its upward scalability.

2.5.2 Possible Optimizations

Checkpointing implementation can be improved by reducing the overhead of the checkpointing process itself, specifically by reducing the amount of state that must be saved and by overlapping the execution of the application with the saving of the state. Concurrent checkpointing relies on the memory protection hardware available in modern computer systems to continue the execution of the process while its checkpoint is being saved on stable storage. The address space is protected from further modification at the start of a checkpoint and the memory pages are saved to disk concurrently with the program execution. If the program attempts to modify a page, it incurs a protection violation. The checkpointing system copies the page into a separate buffer from which it is saved on stable storage. The original page is unprotected and the application program is allowed to resume. Implementations that do not incorporate concurrent checkpointing may pay a heavy performance overhead unless the checkpointing interval is set to a large value, which in turn would increase the time for rollback.

Adding incremental checkpointing to concurrent checkpointing can further reduce the overhead. Incremental checkpointing avoids rewriting portions of the process states that do not change between consecutive checkpoints. It can be implemented by using the dirty-bit of the memory protection hardware or by emulating a dirty-bit in software.

Incremental checkpointing can also be extended over several processes. In this technique, the system saves the computed parity or some function of the memory pages that are modified across several processes. **Error! Reference source not found.** This technique is very similar to parity computation in RAID disk systems. The parity pages can be saved in volatile memory of some other processes thereby avoiding the need to access stable storage. The storage overhead of this method is very low, and it can be adjusted depending on how many failures the system is willing to tolerate.

Another technique for implementing incremental checkpointing is to directly compare the program's state with the previous checkpoint in software, and writing the difference in a new checkpoint. The required storage and computation overhead to perform such a comparison may waste the benefit of incremental checkpointing. Another variation on this technique is to use probabilistic checkpointing. The unit of checkpointing in this scheme is a memory block that is typically much smaller than a memory page. Changes to a memory block are detected by computing a signature and comparing it to the corresponding signature in the previous checkpoint. Probabilistic checkpointing is portable, and has lower storage and computation requirements than required by comparing the checkpoints directly. On the downside, computing a signature to detect changes opens the door for aliasing. This problem occurs when the computed signature does not differ from the corresponding one in the previous checkpoint, even though the associated memory block has changed. In such a situation, the memory block is excluded from the new checkpoint, which therefore becomes erroneous. A probabilistic analysis has shown that the likelihood of aliasing in practice is negligible, but an experimental evaluation has shown that probabilistic checkpointing could be unsafe in practice.

A compiler can be instrumented to generate code that supports checkpointing. The compiled program contains code that decides when and what to checkpoint. The advantage of this technique is that the compiler can decide on the variables that must be saved, therefore avoiding unnecessary data. For example, dead variables within a program are not saved in a checkpoint though they have been modified. Furthermore, the compiler may decide the points during program execution where the amount of state to be saved is small. Despite these promising advantages, there are difficulties with this approach. It is generally undecidable to find the point in program execution most suitable to take a checkpoint. There are, however, several heuristics that can be used. The programmer can provide hints to the compiler about where checkpoints should be inserted or what data variables should be stored. The compiler may also be trained by running the application in an iterative manner and by observing its behavior. The observed behavior could help decide the execution points where it would be appropriate to insert checkpoints. Compiler support could also be simplified in languages that support automatic garbage collection. The execution point after each major garbage collection provides a convenient place to take a checkpoint at a minimum cost.

2.5.3 Current State of the Art

Checkpointing implementations available to production systems tend to be fragile. Operating System-level implementations have been available on some commercial systems, but they tend to be accompanied with a long list of caveats that make it difficult for the programmer to assert whether it can be safe to deploy it for a particular program. Open-source and public-domain implementations that operate at the user-level have also been available, but since they cannot capture the state of the system, they may not be sufficiently robust to handle all applications. Frustrated with this state of affair, most programmers have taken matters into their hands, implementing routines to support application checkpointing specifically designed for their programs. The advantage of this approach is that it allows the programmer to decide what to save and when to save it. Yet, this is also the disadvantage of this method. A mistake by the programmer in implementing the checkpointing process may make it impossible for the application to restart correctly if a critical variable was not saved, or if the checkpointing is not done frequently enough. Also, it requires the programmer to complicate the logic of the program with checkpointing, and understand the MTBF terms of the system, hardly a recipe for portability or robustness. The performance overhead in this user-level checkpointing is also high, for instance, because the programmer has to use the file system interface to write the checkpoints and cannot rely on any system-level optimization of the storage structure and since everything has to be done from the program level, additional layers of overhead make the process more complex and expensive.

3 Future Trends and Impact on Resilience

Several trends in technology are emerging and will affect the construction of future systems. We explore some of the trends and how they will affect system resilience. The purpose of this exploration is to understand how incremental improvement in resilience will be effective in face of these technology trends, and identify areas where investments need to be directed to prepare future systems.

3.1 Power Management

For environmental as well as economical reasons, power consumption of computing systems is a major concern for system designers and users. A full survey of the state of the technology in power management is not in the scope of this report. What is relevant here is how power management will interact with system resilience.

Current and future power management technologies will encompass all system components, including processor cores, memories, storage, I/O circuitry, power supplies, service processors, etc. These technologies include slowing components down, such as Dynamic Voltage and Frequency Scaling (DVFS) of processor and memory chips, slowing down the rotational speed of magnetic disks, and speed control of data communication on external network fiber or network switches. More aggressive power management techniques will also include resource deactivation such as turning off individual cores within a multi-core processor chip, disabling some cache lines and turning off specific memory banks. Turning off disks has been used in mobile systems for years and is finding its way into large-scale server environments. We will also see power supplied to a specific subsystems being capped.

Power management will be performed voluntarily under application control or involuntarily under operating system or even hardware control. For example, power management can be instigated by programmer control by providing hints to the operating system about the parts of the program where processor speed can be reduced (e.g. while waiting for a message). But the programmer may not have ultimate control, for example the hardware may decide to slow the processor down to reduce thermal stresses and hot spots under extreme conditions of workload intensity.

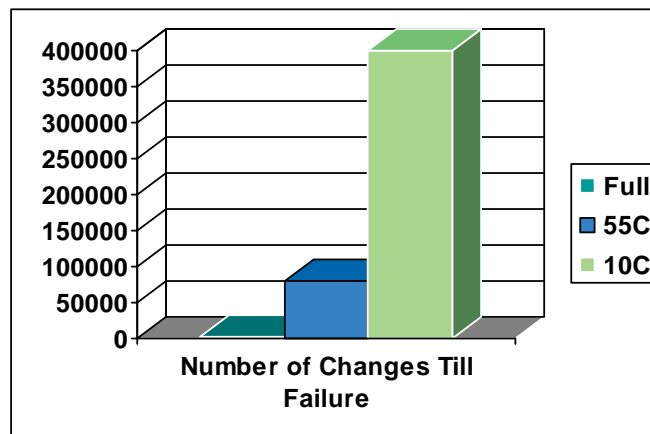


Figure 5 Effects of continuous power on and off of a chip on its longevity.

The interaction of power management with system resilience is likely to be negative. Power management will create thermal variations that will induce mechanical stresses at chip and board levels. Furthermore, the continuous change in the disk rotational speed or the frequent activation and deactivation of disks will create unprecedented reliability challenges to the mechanical components of the disks. For example, enterprise-class disks are designed to sustain a power on-off every 8 hours, certainly less frequent than power management systems are expected to

change the disk speeds. The mechanical stresses, when accumulated over time, may lead to board-level failures in the form of separated or shorted connections. Additionally, the accumulated effects of mechanical and thermal stress will reduce the longevity of individual chips and magnetic disks. Figure 5 shows the expected number of turn on-off cycles as a function of the thermal swing. Notice the order of magnitude reduction in chip longevity when the thermal swing increases from 10C to 55C. The net effect of all the failure acceleration may be the reduction of the expected reliability of the individual components (component Mean Time To Failure, MTTF), which in turn may negatively affect the system's overall MTBF. Alternatively, it may also cause more expensive qualification and components testing, which may drive a large system cost beyond the realm of affordability.

At the large scale, there may be some electrical stresses at the data center level. For example, if power management is applied to reduce the voltage and power of 100,000's of processors simultaneously will create large power swings of megawatts within a few microseconds. We note today that large-scale installations have a routine by which machine bring up and down are cascaded through out the system to avoid these swings.

Another area where our understanding is incomplete is the interaction of power management with soft errors. There will be two opposing effects at play due to the operation at lower voltages. On one hand, the threshold for errors will be reduced and thus tolerance to soft errors at the system level will be reduced. On the other hand, lower power consumption will reduce temperature and white noise, which will counter act the first effect, potentially. Models need to be developed to study the effects of these interactions.

ACCELERATED TESTING

Th Foil Fail Rate Bulk Devices

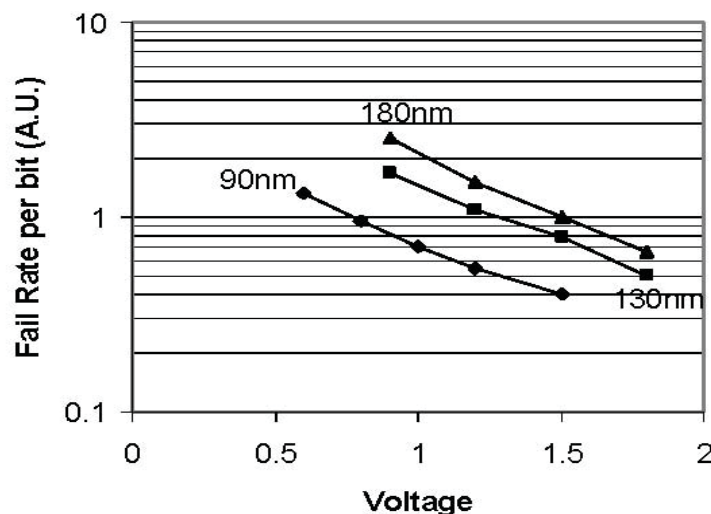


Figure 6 Variation of SER with voltage for 3 Silicon generations.

Power management also may interact in interesting ways with existing software. So far, software has been designed and tested with all components running at the same speed. It is conceivable that with power management changing the speeds of these components, some timing bugs may be uncovered. It is reasonable to argue that software should not have any bugs, but the reality is that functioning software often contains latent bugs that could get exposed if power management starts interacting with the system in ways that were not tested before. At the very least, software testing has to include regression tests that mitigate these effects.

The interactions of power management with resilience, thus, are conjectured to be negative, and we emphasize that no one currently has sufficient experience to assess these interactions and their impact. Also, it is clear that resilience-aware power management algorithms need to be developed at all levels of software and hardware design. We therefore believe that studying the effect of applying power management on the reliability of the system should be an area of research worthy of pursuit.

3.2 System Size

Processor speed improvement due to technology is essentially over. Going forward, Moore's law will allow only denser integration of transistors, which means that systems will grow horizontally. Also, disk speeds are not likely to improve much. Performance demands will force designers to a brute force approach in which more performance will be met by incorporating more components in a system. We already know that the Blue Gene super computer includes about 128,000 processor cores in its maximal configuration. We also know that HPCS-class machines will likely include 100,000's of processor cores, and thus, beyond petascale systems could possibly include millions of processors, memory chips and disks. Assuming the traditional improvement in component-level reliability, the rapid increase in the number of system components will likely reduce the MTBF of the system to a few days or even hours. As discussed in Section 2, the lower MTBF may well force existing techniques for reliability out of consideration. For example, an MTBF of 8 hours may yield an unacceptable overhead of 40% at 10 minutes/checkpoint at the system level. Also, preserving the checkpoint duration may require a non linear increase in the number of disks that are devoted to save the checkpoints, merely to keep up with the checkpoint demands. This may not be even affordable from a financial standpoint. For example, today's system balances are typically 0.5B/F for memory capacity and 0.01 B/F for memory bandwidth. For these balances, the checkpointing time would be about a minute, ideally. System inefficiencies tend to push this figure higher. At any case, the balance appears impossible to sustain at much higher system compute capability. Therefore, it is clear that the current practice *must* be revised at the envisioned level of system sizes for the petascale systems and beyond. New techniques for resilience, possibly impacting the existing practice of writing large-scale programs must be developed.

3.3 Heterogeneity

The studies performed under the HPCS program and others have pointed out the serious limitations of the current model of writing parallel programs based on MPI. Besides its scalability limits, we have pointed out in Section 2 that it has an inherent problem with failure containment, which will likely limit the scalability even further. Recently, we have seen a new trend toward incorporating heterogeneous processors in the system in the form of accelerators,

Field Programmable Gate Arrays (FPGA), and processors of different types. An example of these new systems is the Road Runner system at the Los Alamos National Laboratory.

It is not clear at this point how one can produce a portable technology of providing reliability for these heterogeneous systems. For the current practice of checkpoint/restart, for instance, will require support in capturing the states of the various components. This is not straightforward, for instance, the state of an FPGA may not be captured easily, and some additional support for capturing a meaningful state of the system is needed. Also, application software on these systems will be more complex than the versions that are targeting a single, homogeneous platform. Therefore, it is conceivable that the reliability of the software (operating system, middleware and applications) will reduce the MTBF of the system further. We believe that further research must be directed at developing new algorithms and protocols for building reliable heterogeneous systems. The current status quo is inadequate and will not field the new challenges on the horizon.

4 Plausible Fields of Research

In the preceding sections, we have established that the current practice in building resilient systems will not carry through with the larger systems envisioned at petascale and beyond. New research is needed to develop the necessary technology that will achieve the goal of ensuring acceptable levels of resilience in future systems. In this section, we study potential fields and directions of innovations toward this goal. Some of these directions are inspired by new technology trends that can be exploited to improve system resilience. Others are insights that we have gained by observing what happens in other fields.

4.1 Exploiting New Technology Trends

4.1.1 Exploiting Multi-Core Systems

Future systems will feature processors with many cores. Limitations on memory bandwidth, cache capacity and application software may prevent some of these cores from being put to profitable use all the time. A simple idea is to dedicate some of this processing capability to handle recovery chores and support an increased level of system reliability. This approach seeks to create a different value out of the additional cores other than mere performance.

As an example to illustrate the point, consider the study shown in Figure 7, depicting the performance of four applications on a 4-threaded core. It is clear that there is very little performance improvement that can result from increasing the number of threads from 1 to 2, much less from 2 to 4. In all but one commercial application (JBB), performance hardly improves.

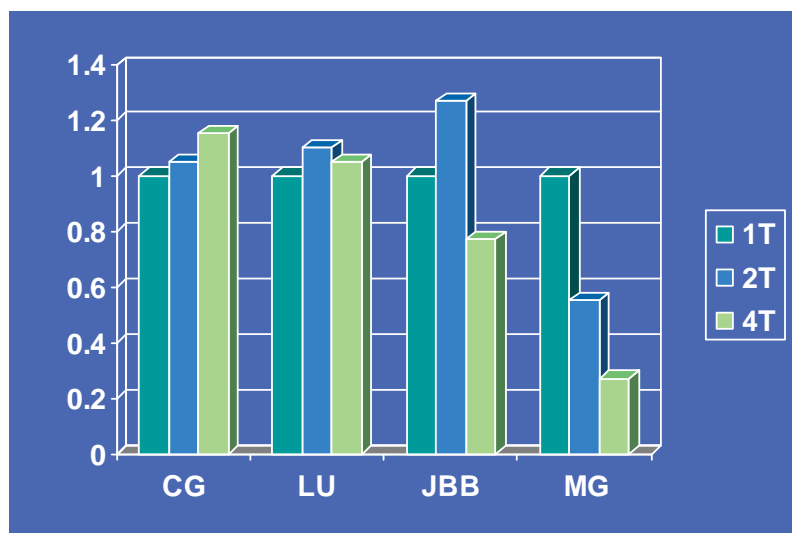


Figure 7 Performance study on a 4-threaded core performance.

Exploiting multi-core systems can take the form of allocating one core (or more) to perform all the monitoring logic that is required by higher level algorithms for providing reliability. For example, Figure 8 shows an example of an 8-core processor chip, in which 2 cores are set aside for resilience support. This support may include diagnosis and error checking, or could be made available to the compiler to include reliability support. How to do this requires a lot more research, but it appears a plausible and promising approach.

Figure 9 shows another potential use of multi-core systems, in which the cores are paired in the modules, each consisting of a primary and a backup. This approach could be used to detect and recover from soft errors, and could yield substantial redesign of the processor logic system. For example, this design allows less emphasis on diagnostic and error checking circuitry in the logic, which could reduce the design complexity and power consumption. Again, how to do this requires further research both in hardware and software (e.g. the operating system must be involved). The approach again appears plausible and promising.

A research agenda can be built around exploiting multicore systems for reliability purposes, including further investigations of the examples provided here and others. There is a need to prototype and explore these possibilities to determine the most effective exploitation of additional cores and threads. System software will also be impacted and it may not be trivial to include such support (e.g. compiler exploitation of multicore for the purpose of reliability). We also need to assess the interactions with power management, and how it affects resilience in such configurations. Finally, this has to be integrated into an overall strategy for providing resilience in future systems.

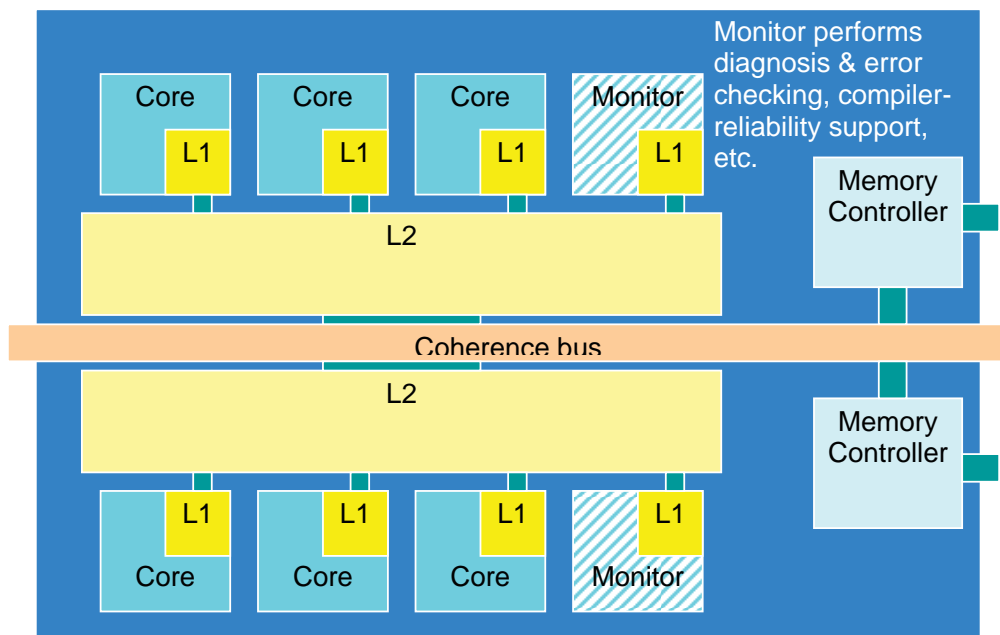


Figure 8 A multicore system with two cores devoted to resilience support.

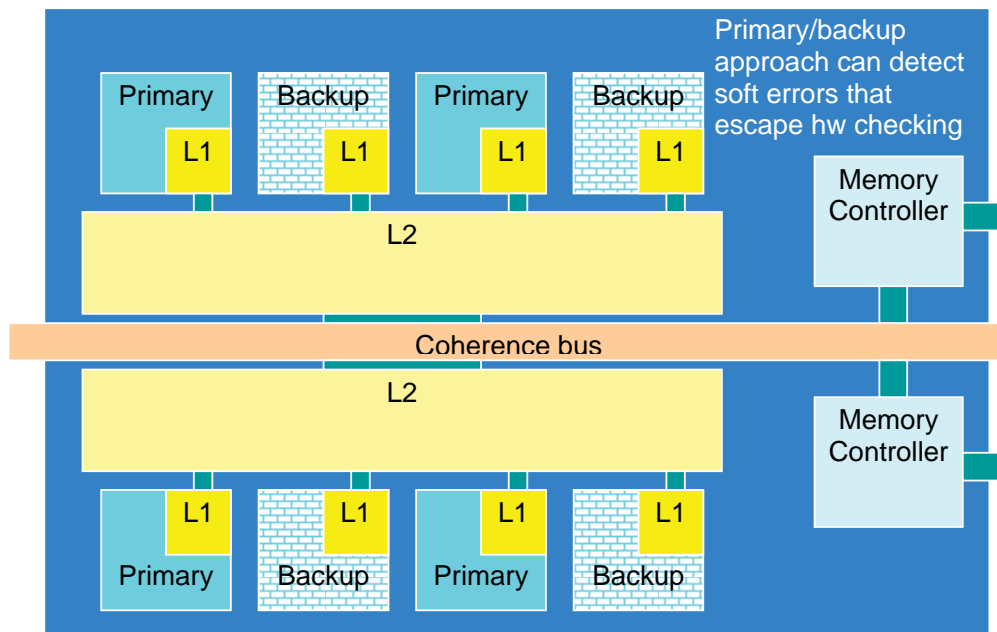


Figure 9 A multicore system with a primary/backup approach.

4.1.2 Exploiting Semiconductor-based Stable Storage

As described in Section 2, a central stable storage facility is required to support resilience and reliability in today's systems. We argued that this approach, however, is not scalable, especially when we consider that future systems may have lower MTBF and therefore a requirement for more frequent checkpointing. We also know that the focus on strong scaling with require at least a 10X reduction in checkpoint latency. These requirements cannot be met by incremental improvement of existing checkpointing techniques and disk-based stable storage. A new trend in technology is the availability of semiconductor-based stable storage, e.g. in the form of flash memory.

Flash memory can be exploited to extend the life of existing checkpointing techniques. For instance, it can be used as a local flash disk to store the checkpoints within each system. Then, when all checkpoints have been captured, which would be much faster than today's approach, the checkpoints are staggered to the disk in a manner that reduce the overall contention on the disks. This may allow existing disk-based checkpointing to continue to be useful while coping with the requirements of increasing the checkpointing scheme. This 2-level approach is shown in Figure 10.

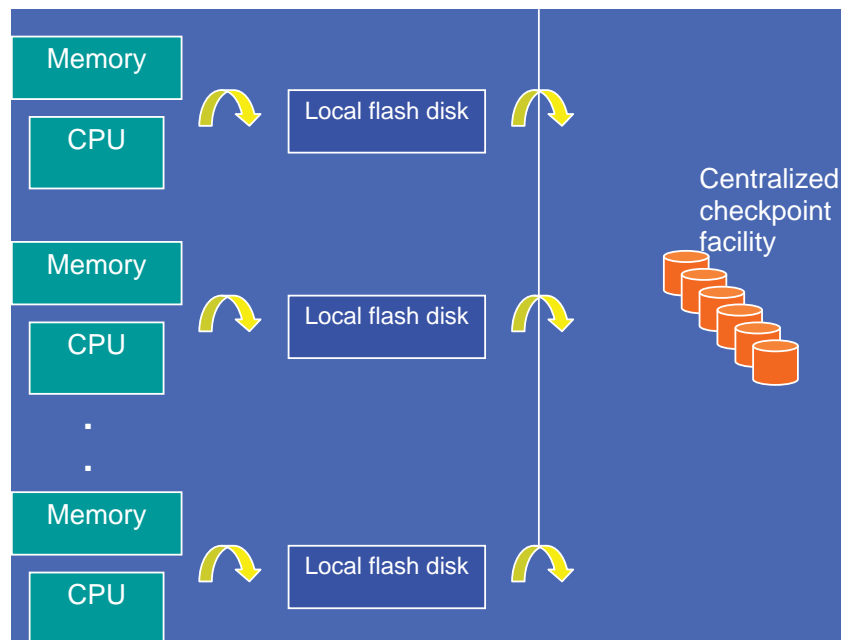


Figure 10 A 2-level checkpointing approach that exploits flash memory.

Another approach is to eliminate disk-based checkpointing altogether, and use the flash memory in a neighboring system instead. Figure 11 shows this proposal. The exploitation of flash memory is certainly a vast area of research, and we have only shown two proposals. Many others are possible and certainly this is one of the most promising approaches going forward.

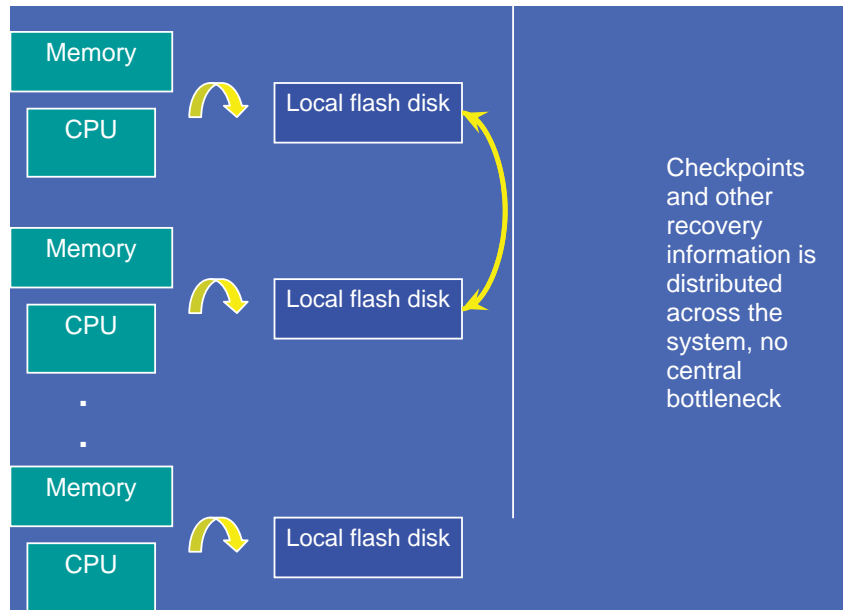


Figure 11 Distributed checkpointing using flash memory.

4.1.3 Exploiting Virtualization

Virtualization is gaining momentum in the commercial domain. Virtualization can be leveraged in high-performance computing systems as it provides a robust state capture and migration. This can be used as a robust building block to implement failure recovery, for example through checkpointing an entire partition. This relieves the application programmer from having to manage reliability and closes the gap in robustness between existing fragile checkpointing techniques and the desired level of robustness.

4.2 New Technologies

4.2.1 New Programming Models

We have established that the flat message passing model based on MPI is at the root of the failure containment problem, which we perceive as the most severe going into petascale systems and beyond. Other programming style are emerging (or re-emerging). For example, what used to be called the “bag of tasks model” in the 1990’s is back under the name of map-reduce. Other models such as the one employed in X10 confines the communication into semantically controlled interactions that could be constrained for the purpose of reliability. These programming models do not suffer from the legacy of MPI, and therefore it is useful to re-examine the current practice of checkpoint restart as the main method to provide resilience in such systems.

Some of these new programming models is inherently fault-tolerant, and thus is more scalable. For example in Figure 12 shows how the map-reduced model can help failure containment. In this model, the tasks communicate through data repository such as a histogram or input files. As a result, if one of the processing nodes fails, it can be restarted on any other system, unlike in MPI system when a single failure requires the entire system to be restarted.

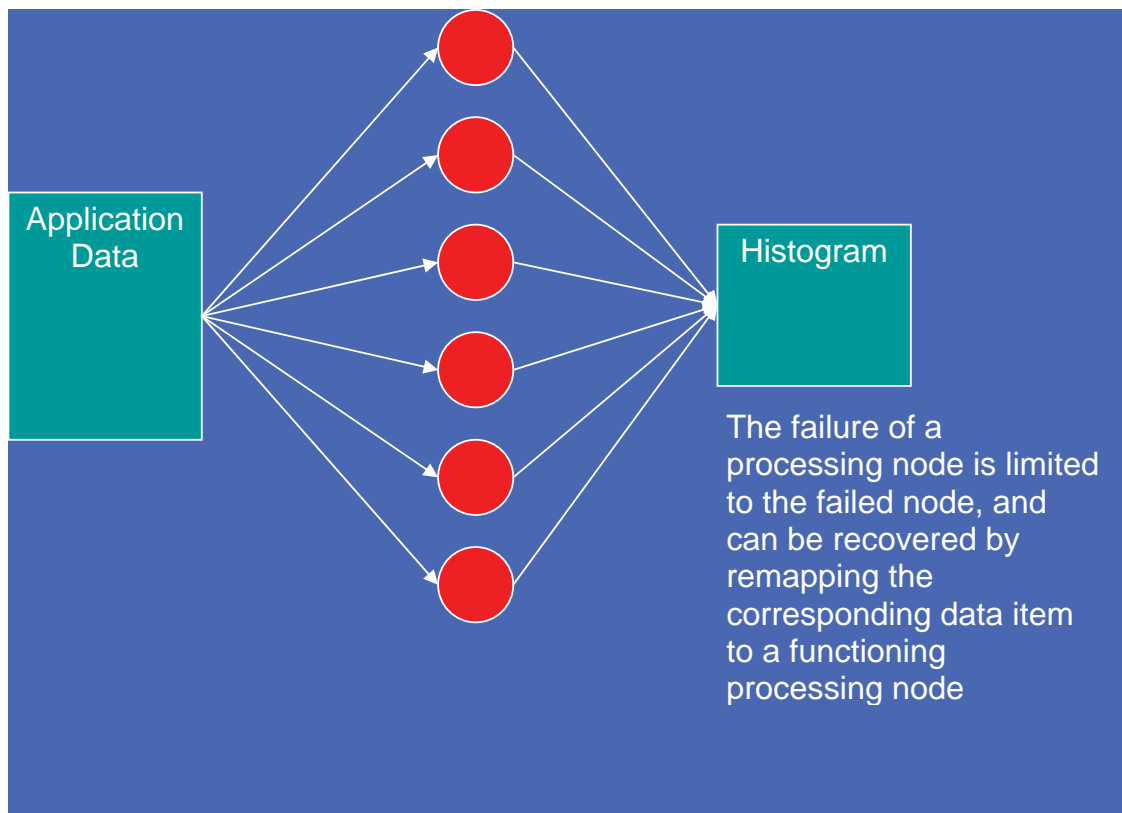


Figure 12 Containment in the map-reduce model.

4.3 Application of Machine Learning

Machine learning techniques use monitoring to build a finger print of the normal system behavior such that over time, any deviation from this normal behavior can be detected and handled. For example, machine learning techniques can be used to sift through the massive amount of data in system logs, which are intended for system administrators but are never read. Also, over time, these statistical machine learning techniques can build fingerprints for different pathological system behaviors. A database of fingerprints can thus be built over time and analyzed continuously to detect and potentially correct (or help in correcting) errors.

4.4 Compiler Support for Resilience

Software remains an important source of failures in high-performance computing systems. Currently, software verification and exhaustive testing are still beyond the capability of today's systems. Also, there are environment-dependent bugs such as memory exhaustion timing anomalies that cause problems that are not detected during normal operation. Bug detection logic inserted into the application by the compiler is an open area of research that has so far not been tapped.

A potential structure of the solution would add modules to automatically patch software while it is operational, modules for error detection and tolerance, including for instance detection of memory leaks, races, and monitoring techniques for anomaly monitoring. Coupled with the abundance of cores that we described in Section 4.1.1, there is a new promising field of research on how to equip code to cope with all sorts of failures in a programmer-transparent manner.

5 Bibliography

1. K. Birman. "How and Why Computer Systems Fail," in *Reliable Distributed Systems*, Springer New York, pp. 237—246.
2. E.N. Elnozahy, L. Alvisi, Y-M. Wang, and D. Johnson. "A Survey of Rollback-Recovery Protocols in Message-Passing Systems." In *ACM Computing Surveys*, vol. 34, Sep 2002.
3. J. Gray and A. Reuter. "Transaction Processing: Concepts and Techniques", Morgan Kaufmann, 1993.
4. B. Schroeder and G. Gibson. "A Large Scale Study of Failures in High-Performance Computing Systems", in Proceedings of the *International Symposium on Dependable Systems and Networks (DSN2006)*.
5. Wikipedia. "Multitier Architecture", the Wikimedia Foundation.

APPENDIX D

FINAL REPORT OF EMBEDDED EXTREME SCALE SYSTEMS STUDY

Technical Report, Contract FA8650-07-C-7724
GT Fund R8251, ECE Project 210667V



Embedded Terascale System Analysis and Design Environment Report

Mark A. Richards, Timothy Scott
School of Electrical and Computer Engineering,
Georgia Institute of Technology

Daniel P. Campbell
Georgia Tech Research Institute, Georgia Institute of Technology

Thomas Conte, Jason Poovey
College of Computing, Georgia Institute of Technology



Submitted to:

U.S. Air Force Research Laboratory
AFRL/SNDI
Bldg. 620, Room 3 DU57
2241 Avionics Circle
Wright-Patterson AFB, OH 45433-7320
ATTN: Ms. Kerry Hill

Submitted by:

GEORGIA INSTITUTE OF TECHNOLOGY
A Unit of the University System of Georgia
Georgia Tech Research Institute
Atlanta, Georgia 30332-0800

Contracting through:

GEORGIA TECH RESEARCH CORPORATION
Centennial Research Building
Georgia Institute of Technology
Atlanta, Georgia 30332

May 2011

**The views expressed are those of the authors and do not reflect the
official policy or position of the Department of Defense or the U.S. Government.**

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

EES Report

TABLE OF CONTENTS

<u>Paragraph</u>	<u>Page</u>
PREFACE VIII	
1. INTRODUCTION	1
1.1 DARPA's Extreme Scale Computing Studies	1
1.2 The Embedded Extreme Scale Study	1
2. PREDICTABLE DESIGN OF EMBEDDED EXTREME SCALE COMPUTING SYSTEMS.....	3
2.1 Introduction	3
2.2 Some Common Processor Design Approaches.....	3
2.2.1 Incremental System Design.....	3
2.2.2 Analytical Performance Approximation.....	3
2.2.3 Simulation-Based Design	4
2.3 Predictable Design	6
2.3.1 Sources of Unpredictability.....	7
2.3.2 Simulation-Based Uncertainties	8
2.3.3 Architectural-Based Uncertainties	9
2.4 Overview of Statistical Sampling.....	9
2.4.1 Confidence Intervals	10
2.4.2 Types of Statistical Sampling.....	10
2.5 Summary: Predictable Design.....	13
3. APPLICATION ANALYSIS AND BENCHMARKS FOR EMBEDDED EXTREME SCALE SYSTEMS.....	14
3.1 Introduction	14
3.2 Computational Requirements for Examples of Embedded Extreme-Scale Computing Applications	15
3.2.1 KAPE.....	15
3.2.2 Persistent Wide-Area Radar Surveillance.....	20
3.2.3 Streaming Sensor Challenge Problem Development	22
3.2.4 Autonomous Vehicles.....	27
4. DATA MOTION ANALYSIS FOR LOW POWER ALGORITHMS.....	34
4.1 Locality Analysis of Algorithms	34
4.1.1 Locality Metrics	34
4.1.2 Locality Analysis Tools.....	37
4.1.3 Spatial and Temporal Locality Experiments	39
4.2 Data Motion Metric.....	40
4.3 Data Motion Analysis Conclusions	41

TABLE OF CONTENTS

<u>Paragraph</u>	<u>Page</u>
5. HIGH PERFORMANCE LIBRARIES FOR MULTICORE PROCESSORS	43
5.1 Background.....	43
5.2 VSIPL	44
5.3 GPU VSIPL.....	45
5.4 Progress in the Development of GPU VSIPL.....	46
6. METRICS FOR EXTREME SCALE SYSTEMS.....	47
6.1 Introduction	47
6.2 Measuring Programmability	48
6.2.1 Cognitive Dimensions and Programmability.....	48
6.3 Parallel Programming Patterns.....	49
6.3.1 Standard Parallel Problems.....	50
6.4 Proposed Methodology.....	51
6.5 Experiments	53
6.5.1 Example 1 - Quicksort.....	54
6.5.2 Example 2 - MapReduce.....	55
6.6 Programmability Metric Conclusions.....	57
REFERENCES.....	58

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
Figure 1. Hierarchical design levels and corresponding simulation levels in the RASSP program. After [6].	5
Figure 2: As performance increases, the a priori ability to predict the performance before the design is built decreases.....	6
Figure 3: Cluster sampling for multiprocessor architectures.	12
Figure 4. General structure of an embedded streaming sensor signal processing system.....	14
Figure 5. Functional overview of KAPE detection processor.	16
Figure 6. Example of KAPE detection performance improvement. From [23].	17
Figure 7. Top-level SAR-based persistent WAS processing flow.....	21
Figure 8. Processing flow for optional digital spotlighting step.	21
Figure 9. Backprojection algorithm.	22
Figure 10. Top-level SSCP processing flow.	23
Figure 11. Affine registration processing flow.....	25
Figure 12. Thin spline registration processing flow.....	26
Figure 13. GT's Sting autonomous vehicle.	28
Figure 14. Sting hardware architecture.....	29
Figure 15. Sting software architecture.....	30
Figure 16. Major steps in the LOIS lane tracking algorithm.	30
Figure 17. Major steps in the stopline algorithm.	31
Figure 18. Major steps in the (a) mapper, (b) planner, and (c) controller algorithms.	32
Figure 19. Major steps in the robust lane detection algorithm.	33
Figure 20. Abstract memory hierarchy for binning locality scores.....	36
Figure 21. "Binned" temporal locality L_T vs. trace length for a basic matrix multiplication algorithm.....	37
Figure 22. Temporal and spatial locality vs. algorithm size.	39
Figure 23. Data motion metric for several numerical algorithms.	41
Figure 24. Growth of GPU single precision floating point performance.	43
Figure 25. Output of GPU VSIPL range-Doppler map application showing three targets with sidelobes in both range (vertical) and Doppler (horizontal) dimensions.....	45
Figure 26. OPL 2.0 Patterns.....	49

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
Figure 27. "Temperature chart" indicating reliance of various application codes on the 13 canonical parallel programs. From [53].	51
Figure 28. Cognitive dimension scores for two implementations of the Quicksort algorithm.	55
Figure 29. Programmability score for two Quicksort implementations.	55
Figure 30. Cognitive dimension scores for two implementations of the MapReduce algorithm.	56
Figure 31. Programmability score for two MapReduce implementations.	57

LIST OF TABLES

<u>Table</u>	<u>Page</u>
Table 1. KAPE System Parameters for Various Operational Modes	18
Table 2. KAPE Systems Computational Summary.	19
Table 3. Computational load for SAR-based persistent WAS image formation.	21
Table 4. Tentative Challenge Problem Scenarios.....	26
Table 5. Scenario Loadings.	27
Table 6. Computational Style of Major Functions.	27
Table 7. Estimated computational requirements of the LOIS lane tracking algorithms.....	30
Table 8. Estimated computational requirements of the stopline algorithm.	31
Table 9. Example of spatial locality stride calculation with W=4.	35
Table 10. Example of reuse distance calculation.....	36
Table 11. Cognitive Dimensions	48
Table 12. Thirteen Canonical Algorithms.....	50
Table 13. Potential for Automated Measurement.....	53
Table 14. Measurements for Quicksort program.....	54
Table 15. Measurements for MapReduce program.....	56

LIST OF ACRONYMS

AFRL	Air Force Research Laboratory
API	Application Programming Interface
CCD	Coherent Change Detection
CFAR	Constant False Alarm Rate
CP	Challenge Problem
DARPA	Defense Advanced Research Projects Agency
DFT	Discrete Fourier Transform
DSM	Distributed shared Memory
ECS	Exascale Computing Study
ECRS	Exascale Computing Resiliency Study
ECSS	Exascale Computing Software Study
EES	Embedded Extreme Scale
Eflops	Exa floating point operations per second (10^{18} flops)
EMI	Electromagnetic Interference
EO/IR	Electro-Optic/Infrared
flops	Floating point Operations Per Second
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
fps	Frames Per Second
Gbps	Giga bits per second (10^9 bps)
Gflops	Giga floating point operations per second (10^9 flops)
Gops	Giga operations per second (10^9 ops)
GPU	Graphical Processing Unit
GT	Georgia Tech
GTRI	Georgia Tech Research Institute
HCI	Human-Computer Interface
HPEC-SI	High Performance Embedded Computing Software Initiative
HPCS	High Productivity Computing Systems
IPC	Instructions Per Clock Cycle

LIST OF ACRONYMS

KAPE	Knowledge-Aided Parameter Estimation
KASSPER	Knowledge-Aided Sensor Signal Processing and Expert Reasoning
LRU	Least Recently Used
Mops	Million operations per second (10^6 ops)
OPL	Our Pattern Language
Pflops	Peta floating point operations per second (10^{15} flops)
QRD	Q-R Decomposition
RAMP	Research Accelerator for Multiple Processors
RASSP	Rapid Prototyping of Application-Specific Signal Processors
RF	Radio Frequency
ROI	Region of Interest
SAR	Synthetic Aperture Radar
SBD	Simulation-Based Design
SLAM	Simultaneous Location and Mapping
SSCP	Streaming Sensor Challenge Problem
STAP	Space-Time Adaptive Processing
SVD	Singular Value Decomposition
SVM	Support Vector Machine
TASP	Tactical Advanced Signal Processor
Tflops	Tera floating point operations per second (10^{12} flops)
UAV	Unmanned aerial vehicle
UCB	University of California, Berkeley
UHPC	Ubiquitous High Performance Computing
VHDL	VHSIC Hardware Design Language
VHSIC	Very High Speed Integrated Circuit
VS IPL	Vector, Signal, Image Processing Library
WAS	Wide Area Surveillance

PREFACE

This document is a Technical Report under Georgia Institute of Technology (Georgia Tech, GT) Project 210667V, "Exascale Computing Study", but focused specifically on the portion of the project identified as "Characterization and Design of Embedded Terascale Computing Systems Based on Exascale Computing Technology". This project was sponsored by the Information Processing Technology Office (IPTO) of the US Defense Advanced Research Projects Agency (DARPA) and was administered by the U.S. Air Force Research Laboratory under contract FA8650-07-C-7724.

The authors would like to thank Dr. William Harrod of DARPA/IPTO, the Exascale Computing Study program manager, and Ms. Kerry Hill of AFRL for their support of this effort.

SECTION 1

INTRODUCTION

1.1 DARPA's Extreme Scale Computing Studies

The U.S. Defense Advanced Research Projects Agency (DARPA) [1] is conducting a series of studies under the umbrella of the "Exascale Computing Study". An exascale computer system is a computer system with approximately a one thousand-fold increase in capabilities relative to the computer systems currently under development by DARPA's High Productivity Computing Systems (HPCS) program [2]. In 2007-2008, DARPA conducted an Exascale Computing Study (ECS) addressing hardware and system architecture issues in the development of exascale computing systems. The ECS identified the development of effective parallel programming methodologies for systems having extreme degrees of concurrency, and exascale system resiliency, as critical issues. The final report of the ECS has been publicly released [3].

DARPA then initiated two supplemental studies to further define the technologies and investments needed by 2010 to enable the development of exascale computing systems by 2015, the Exascale Computing Software Study (ECSS) and the Exascale Computing Resiliency Study (ECRS). The goal of the ECSS study, which is finalizing its report at this writing, is to identify the key software technologies, approaches, and methodologies that must be in place to effectively utilize the extreme levels of concurrency expected in exascale computing technology. The goal of the ECRS study was to identify the key technologies, approaches, and methodologies that must be in place to ensure scalability of mean time to failure (MTTF) in exascale systems. Both the ECSS and ECRS shared the additional goal of specifying the research problems that must be solved so that the key technologies identified can be in place in time to support initial deployment of exascale technology in 2015. The final report of the ECSS has been publicly released [4]. A "white paper" summarizing the ECRS conclusions is publicly available from the DARPA web site at this writing [5].

From the beginning of these studies, the exascale studies have considered not only true exascale data center class systems, but also smaller-scale systems that would be enabled by the development of exascale technology. Specifically, it is assumed that "departmental" systems, physically on the order of one or two cabinets, could be constructed that operate at petascale levels, and that "embedded" systems, typically at the VME chassis physical scale, could be constructed that operate at terascale levels. The moniker *extreme scale systems* has been adopted to refer collectively to this range of computing capabilities and physical sizes.

1.2 The Embedded Extreme Scale Study

Georgia Tech (GT) serves as the prime contractor for the organization and conduct of these studies, as well as a participant in them. In 2008, DARPA added a task funding GT to study several key issues in realizing embedded extreme scale (EES) computing systems. The EES study comprised three principal sub-tasks. The first sub-task, analysis of terascale embedded applications, focused on identifying embedded computing applications projected to require terascale computing capability in the 2015 time frame. GT analyzed the computational,

communication, memory, and other requirements of the selected applications. Key functional kernels and application benchmarks representative of the selected applications were identified.

The second sub-task focused on a design environment for terascale embedded computing. GT assessed current performance modeling and prediction methods applicable to embedded computing systems. Based on this review, GT proposed new approaches for the hardware and software architecture of a design environment for terascale embedded computing systems.

The third sub-task considered new concepts for inherently low-power computational algorithms. Under this part of the effort, GT investigated techniques for developing new algorithms and improvements to existing algorithms that jointly optimize the energy and runtime required for computation of selected functional kernels.

Sections 2 through 4 of this report describe the research conducted and results obtained under each of these subtasks, respectively. Section 5 describes additional related work in the development of high performance computational libraries for advanced graphical processing units (GPUs).

SECTION 2

PREDICTABLE DESIGN OF EMBEDDED EXTREME SCALE COMPUTING SYSTEMS

2.1 Introduction

Embedded extreme scale computing systems (EES) and their associated software require tens of terascale performance in a chassis-sized package. This presents major design challenges. Chief among these problems is that tight constraints limit the potential for high performance. It is a maxim today that if a design has tight constraints, the ability of a designer to find a design that achieves high performance is compromised. It is beyond argument that embedded chassis-sized systems are tightly constrained. The required performance of tens of teraflops makes finding a design particularly difficult.

2.2 Some Common Processor Design Approaches

There are several known techniques for developing embedded computer designs. The major approaches can be grouped into incremental system design; analytical performance approximation; and simulation-based design. Each of these techniques suffers from different drawbacks. In general, a desirable property of a design approach is that it generate designs whose ultimate performance, when built, matches the predicted performance derived *a priori*. This property is referred to herein as *predictable design*. While this may seem a bare minimum requirement for any design method, in practice it is difficult to achieve.

2.2.1 Incremental System Design

Rarely is a new processor designed entirely from a clean sheet of paper. Instead, incremental system design is very commonly used to design EES systems. In this approach, designers adapt an existing design to new requirements by a process of incremental updates. The existing design is analyzed for bottlenecks. The faulty design aspects responsible for the bottlenecks are then repaired or replaced. That the resulting system performs at a higher level than the original system cannot be denied. However, this incremental approach is similar in theoretical optimization to finding a local minima or maxima. Other designs that may produce revolutionary EES systems are not explored. Often this approach results in a faulty conclusion that it is impossible to improve a particular system beyond a given physical, process or algorithmic bottleneck.

2.2.2 Analytical Performance Approximation

Analytical performance approximation is a technique wherein the potential systems are modeled using approximate mathematical techniques such as Petri nets or queuing system models. Because of the complexity of these models, analytical solutions are often intractable without some standard assumptions about the processor workload (e.g., a Poisson-distributed random process, a balanced birth-death process, etc.). These assumptions are the crux of the drawback with the analytical approach: real workloads do not behave as random processes. Furthermore, it is often the exceptional (rare) workload case that limits the system's overall

capacity and performance, not the common case. The impact of these stressing workloads is difficult to capture with standard analytical approximations.

2.2.3 Simulation-Based Design

Simulation-based design (SBD) refers here to comprehensive modeling of potential systems using event-based or time-step-based modeling techniques to write simulation programs. These simulators take in stimulus recorded from actual running workloads and create a “virtual environment” in which to process these workloads. The advantages of this approach are numerous. Simulation-based design uses engineering science to simulate several candidate systems, selecting from among the candidates based on performance criteria. The performance estimates can be highly accurate. Workloads are exactly modeled because every action of a particular workload is taken into account.

SBD has some drawbacks, however. Architectures that require significant porting of applications may require too much effort to simulate, especially given the likelihood that the ported application effort will be wasted if the architecture does not meet performance goals. However, if automated porting tools are used (i.e., via auto-tuning), this hurdle can be overcome. The most daunting limitation of SBD, though, is the simulation speed. A simulator simulating a system that performs 100 billion operations per second will typically run many of orders of magnitude slower, for example, at only 1 million operations per second. Simulation of long designs may stretch into weeks or months, even on the fastest hardware available. Long simulation times can be reduced by increasing the granularity of the system. For example, task interactions could be simulated instead of individual instruction interactions. The drawback of this acceleration approach is that it necessarily leads to less accurate performance predictions.

An extension of the idea of granularity/accuracy tradeoffs to control simulation time is the idea of hierarchical simulation. This idea was explored extensively in DARPA’s Rapid Prototyping of Application-Specific Signal Processors (RASSP) program in the mid-1990s [6]. The RASSP program advocated a detailed, multi-level hierarchical design approach with the design levels and corresponding simulation and prototyping levels shown in Figure 1. While RASSP relied entirely on VHDL simulation, the concepts are applicable to any simulation environment.

The RASSP hierarchical design and simulation environment was closely tied to an iterative-refinement “spiral” design methodology applied to both hardware and software in a co-design methodology [7], as well as to an extension of the Gajski-Kuhn “Y chart” division of simulation into functional, behavioral, architectural, structural, and physical levels of specificity [8]. The concept is to have relatively coarse but fast simulation accuracy at the higher levels, refining the accuracy (and therefore increasing simulation time) in successive iterations as one works down the hierarchy to more detailed design levels. However, with good design practices, it is possible to reduce the amount of the system that must be simulated at slower run speeds at each level, so that total simulation time becomes manageable.

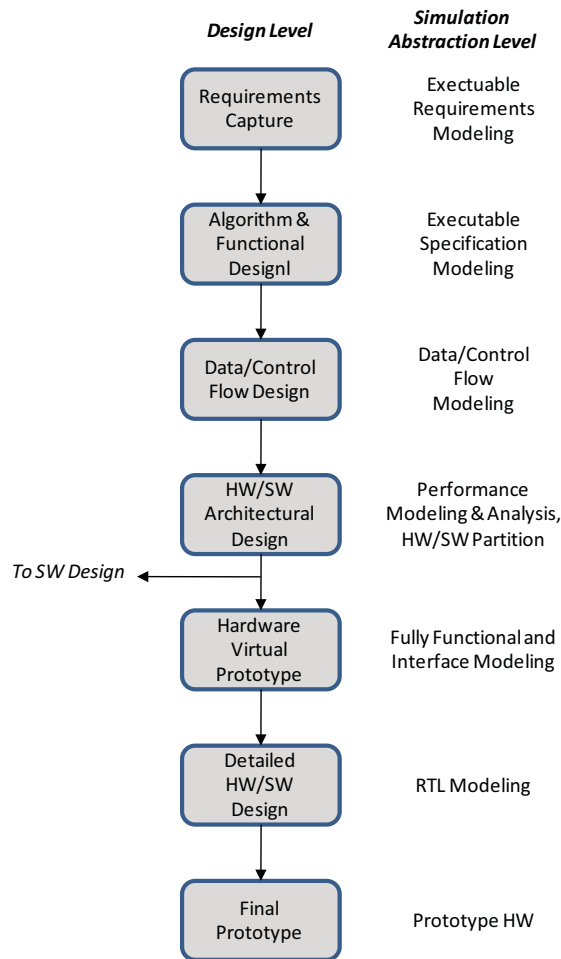


Figure 1. Hierarchical design levels and corresponding simulation levels in the RASSP program.
After [6].

Excessive SBD simulation times can also be mitigated through various simulation acceleration techniques. One approach is the use of parallel computers for simulation acceleration. Another acceleration approach is the use of *emulators*, special-purpose computers, often based on field-programmable gate array (FPGA) technology, to accelerate time-consuming portions of a simulation with good hardware fidelity. A currently-popular example of this approach is the Berkeley “Research Accelerator for Multiple Processors” (RAMP) emulator [9], developed primarily for simulating multiprocessor-based systems. Emulators have several advantages over software simulation. They can be mapped relatively quickly; can be placed in a real environment, and are typically substantially faster than software simulation. However, they are expensive to develop and own, and are typically a limited resource as a result.

2.3 Predictable Design

Predictable design is the property wherein a system's performance is predictable to within some certain acceptable error margin, before the construction of the system. Another way to state this is that it is particularly desirable to have a design whose ultimate performance is predictable *a priori*. This is not a given. Many techniques to accelerate computer architecture performance, such as the use of caches and branch prediction, result in an unpredictable ultimate performance. Stated another way, given an application, the goal is to find a design optimized for a set of performance criteria subject to a set of constraints (power, form factor, etc.). From an optimized design and an application, the goal is to accurately predict performance. Any performance prediction scheme short of custom prototyping (i.e., building the thing) introduces uncertainty.

Figure 2 illustrates the issue and importance of predictable design. The x-axis is the predictability *a priori* of the performance before the design is reduced to practice. A low value of predictability means the design must be actually prototyped to know its true performance; a high value represents a design whose performance can be accurately predicted prior to prototyping. The y-axis represents the observed performance for a particular design approach. Each curve represents a set of implementation methods. The lowest curve ("A") represents a typical tradeoff in the design of systems today. Many aggressive, speculative and dynamic techniques are used to achieve high performance. Although these are useful techniques, their dynamic nature introduces significant unpredictability in the performance of the ultimate system before it is built.

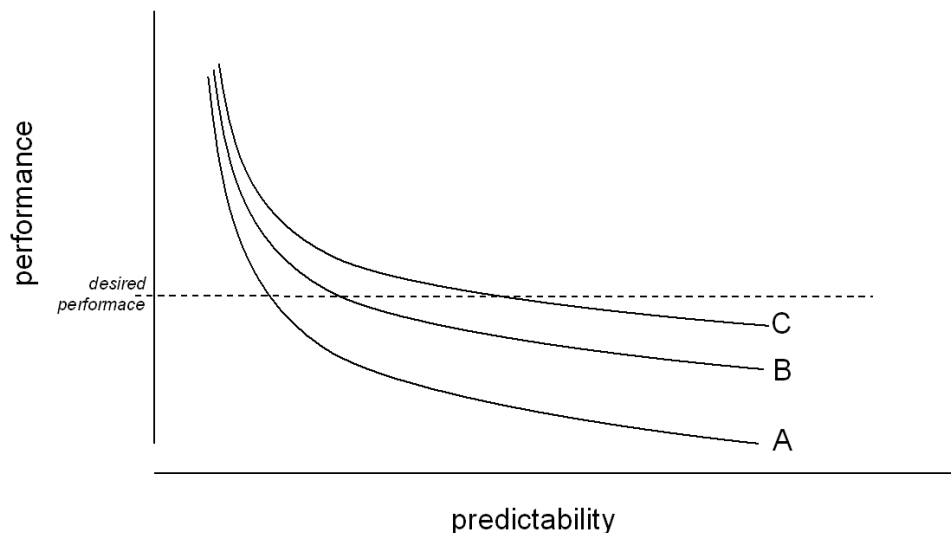


Figure 2: As performance increases, the *a priori* ability to predict the performance before the design is built decreases.

The goal of predictable design is to find design and implementation technologies that can be used to achieve high performance yet yield flatter performance vs. predictability curves. If the horizontal dashed line represents a desired level of performance, the higher curve “B” in Figure 2 constitutes a more desirable set of high performance techniques than does the “A” set of techniques. Design “B” can be thought of as having fewer “unpleasant surprises” than design “A”. Similarly, technique set “C” is even more desirable than “B.”

2.3.1 Sources of Unpredictability

As Figure 2 illustrates, the current techniques for accelerating the performance of EES systems lead to designs that may miss the target performance goal. There are two general reasons for this unpredictability: (1) reliance on flawed simulation results, and (2) dynamically sensitive architectural tricks.

Simulation of EES systems is an arduous task that takes a considerable amount of compute resources. There exist few compute platforms up to the task of simulating future high-performance EES systems in real time (or even tractable time). As discussed above, the granularity of the simulation can be traded for simulation speed, but at a penalty of reduced accuracy. Simulation uncertainties are discussed in more detail in the next section.

Architectural tricks that accelerate a design exploit one of two common properties of workloads: they are predictable in nature, and they exhibit inherent parallelism. Prediction and parallelism form the basis of most architectural “tricks.” For example, caches work well because the memory reference behavior of a section of a workload is self-similar. Thus caches predict the future re-references to memory based on the past. Other prediction techniques include branch prediction, data and instruction prefetching, and, although generally less reliable, prediction of computation results themselves.

Parallelism occurs in any design, often at multiple levels (task, data, and instruction) because there is rarely only one computational result being carried out in a workload. A matrix multiply of two $N \times N$ matrices involves independent calculations of N^2 results, for example. Amdahl’s law predicts that for any time fraction f of a computation that is parallelizable, the maximum speedup S_{\max} through parallelism (corresponding to $P \rightarrow \infty$ parallel processors) can be no more than

$$S_{\max} = \frac{1}{1-f} \quad (1)$$

One aspect of parallelism as a speedup mechanism that is distinct from prediction is that it produces predictable performance gains when the fraction, f , can be measured before runtime. However, by considering dS_{\max}/df , it is easy to see that, for a given uncertainty in the parallelizable fraction, Δf , the resulting uncertainty in the speedup is

$$\Delta S_{\max} = \frac{1}{(1-f)^2} \Delta f = S_{\max}^2 \Delta f \quad (2)$$

Thus, increased reliance on parallelism for speedup also creates an increased sensitivity to errors in estimating speedup when attempting to assess design predictability. Put another way,

maintaining a given level of predictability when increasing parallelism requires increasing accuracy of speedup estimation.

2.3.2 Simulation-Based Uncertainties

Simulation is perhaps the most reliable way to design an EES system. Simulation-based design uses engineering science to simulate several candidate systems, selecting from among the candidates based on performance criteria. There are many challenges to this approach. For example, the system must be simulated performing a given workload (i.e., an application, benchmark or kernel). Architectures that require significant porting of workloads may require too much effort to simulate, especially given the likelihood that the ported workload effort will be wasted if the architecture does not meet performance goals. However, if automated porting tools are used (e.g., auto-tuning compilation systems), this hurdle can be scaled.

Another potentially deadly limitation of simulating systems is poor performance of the simulation itself, as opposed to the system being simulated, as discussed above. This “meta-performance” can readily become a bottleneck to exploring more than a handful of potential designs. The meta-performance problem is due to the complexity of the design simulations: one simulated cycle of the potential future architecture may take several million real-world simulator cycles to simulate with sufficient accuracy. There are three approaches to mitigating this problem: accelerating the simulation algorithmically, exploiting parallelism in the simulation, and trading off simulation accuracy for simulator meta-performance.

There have been several approaches to accelerating computer-system simulation algorithmically. One of the most famous is the use of the least-recently-used (LRU) inclusion property to simulate an entire design space of cache memories with one simulation. It is well-known that certain cache or paging-system replacement policies follow a priority scheme that operates as a stack. These so-called “stacking algorithms” can be simulated in one pass by explicitly modeling the priority stack. In the case of LRU, for example, new data not present in the stack are pushed onto the stack, whereas existing data may be found at a given stack depth. When it is found at a given stack depth, it is “re-pushed” to the top of the stack. This allows for the simulation of an entire space of caches by declaring data found at stack depths beyond the capacity of the hardware cache as “misses,” and those above this critical depth as “hits.” Limited associativity and even direct-mapped cache policies can be simulated using these approaches. A related approach is to use grain-size of the simulation selectively: simulating the most critical aspects of a system in great detail, while simulating others at a very coarse level.

Unfortunately, although there are single-pass algorithms for caches that fully account for hardware configurations, the same is not true for hardware techniques that execute code in parallel. The performance of code in a parallel EES system is a function of many hardware parameters that have a “brittle” impact on ultimate performance. Thus changing just one of these parameters slightly, such as the bandwidth of the interconnect between processing elements, can greatly impact overall performance. Techniques to accelerate the simulation of parallelism are an open topic subject to much active research.

One self-evident approach to speeding up the simulation of parallel computing is to execute the simulator itself as a set of parallel threads or “logical processors.” This has many advantages, but the simulation speedup is modest because the degree of parallelism only has a small

multiplicative aspect on the meta-performance of the simulator. This occurs because the extensive interaction between the simulated elements of the processor being designed requires both large amounts of communication and frequent synchronization between the simulated elements. The overhead in communication and synchronization required for coordinating a high-performance simulation is significant and quickly erases performance gains as the number of threads in the parallel simulation is increased.

The most common technique for speeding up simulation of EES systems is through *statistical sampling*. This process, and its implications for predictable design, is considered in detail in the next section.

2.3.3 Architectural-Based Uncertainties

Computer architecture is, in some senses, a study of “tricks” to speed up a computer. Some of these tricks are done behind the scenes and without the knowledge of the programmer. For example, when code is executed on a pipelined uniprocessor, a branch instruction causes a pipeline stall that robs the overall system of performance. To address this, these control instructions (branches) are predicted at run time and a particular path in the code is speculatively executed. The performance reduction is now a direct function of how accurately the predictor can guess the performance of the branches. The problem with this approach is subtle: the behavior of the system cannot be accurately guessed *a priori* without running the workload through a very detailed simulation.

However, there are other approaches to the same class of problems. In the above example, code from beyond the branch can be moved before the branch to reduce the branch’s impact on performance, provided the code is independent of the branch. This approach is often carried out in the compiler, and is invisible to the programmer. The advantage of this approach is that the dependence of the performance prediction on the program’s dynamic behavior is greatly reduced. A similar and related approach is to use predicated execution to turn conditional code into straight-line code [10].

Another contrast can be drawn in the realm of explicitly parallel workloads. Most programming paradigms fall within one of two large classes: shared memory or message passing paradigms. In the shared memory approach, such as a DSM (distributed shared memory, the OpenMP programming model) machine, the hardware is responsible for transferring shared data from one computational element (processor) to another. This is done via coherence techniques, for example. The DSM approach works well, but it provides a high level of variability in performance due to variations in the actual run-time behavior of the workload. In contrast, a message passing approach (the MPI programming model) leaves it to the programmer to explicitly move data from where it is calculated to where it is needed. Although this puts more work on the programmer, the performance of resultant code is much easier to predict *a priori*.

2.4 Overview of Statistical Sampling

Statistical sampling relies on statistical techniques to reduce the number of events (instructions, messages, etc.) that must be simulated in order to produce accurate predictions of system behavior. An important advantage of this method is that, when used to accelerate simulation, it also allows for the prediction of the confidence (error) in the result, which ultimately leads to more predictable designs. If uncertainty is not bounded, even if a simulation could be

performed instantaneously, its results would be meaninglessly random. The goal in accelerating simulation is to sacrifice a *measurable* amount of accuracy in order to accelerate simulation. Without observing this, there is no confidence in the results. By using statistical theory, the uncertainty of a measurement can be bounded without having to simulate every event of the workload.

2.4.1 Confidence Intervals

In parametric statistics, which rely upon a normal distribution, one measure of uncertainty is the *confidence interval* *CI*. The confidence interval defines a range of values wherein the true mean of a parameter is expected to be contained. In order to calculate a confidence interval, a *confidence level* must first be selected. The confidence level is the probability with which the true mean would be expected to be within the confidence interval. It is common for scientific and engineering studies to select a 95% confidence interval. The reason for this is that 99% confidence intervals are too broad (e.g., “it is known with 100% confidence that the true mean is between 0 and infinity”). On the other hand, 90% confidence intervals are unnecessarily conservative in their estimations of error bounds while also requiring significantly more simulation effort for marginally small confidence gains.

Given a sample $x_i, i = 1, \dots, n$ of measured data, the standard deviation s and sample mean \bar{x} are

$$\bar{x} = \sum_i x_i, \quad s = \sqrt{\frac{\sum_i (x_i - \bar{x})^2}{n-1}} \quad (3)$$

Assuming normal statistics, the 95% confidence interval is then

$$CI (95\%) = \bar{x} \pm 1.96 \frac{s}{\sqrt{n}} \quad (4)$$

It has been often claimed that in computer architecture, the parameter(s) of interest will not be normally distributed, thereby violating the assumption of normality. This would require resorting to nonparametric tests. However, the Central Limit Theorem may be leveraged if one only cares about average metric performance.¹ Since nearly all computer architecture metrics of interest are such summary metrics (e.g., instructions per clock cycle, or IPC), the assumption of normality can be assumed to hold and Student-t statistics can be applied. Student-t statistics compare the distribution of a sampled population to a known distribution and calculate the probability that the two distributions are different.

2.4.2 Types of Statistical Sampling

There are three general techniques of statistical sampling: *simple random sampling*, *stratified sampling*, and *cluster sampling*. Simple random sampling (i.e., sampling individual instructions) is cost-prohibitive for processor simulations, so researchers have typically relied on either cluster sampling or stratified sampling [11],[12],[13]. Stratified sampling refers to the process of

¹ The Central Limit Theorem states that the distribution of metric averages will be normally distributed if the number of data points is sufficiently large (typically, $n > 30$).

sampling where the population (e.g., the memory accesses) is divided into non-overlapping groups, or strata. For example, simulating only particular sets in a cache would be stratified sampling [12]. Stratified sampling focuses on a subpopulation, and predicts that this population behaves the same as the whole. Once the strata have been defined, then each stratum is randomly sampled (e.g., each selected set in the cache is randomly sampled). All elements of the population must be included into one (and only one) stratum, and the strata should generally be homogeneous in nature. However, stratified sampling requires accurate information about the population being sampled in order to classify the population elements. For processor simulation, the act of profiling the instruction stream in order to define the stratum is expensive, and so stratified sampling is not widely used. However, in multiprocessor simulations, stratified sampling is possible by picking a subset of the cores as the strata to sample. Tractable approximations to stratified sampling techniques have been proposed in [14].

Cluster sampling is typically used when “natural” groups are evident in the population. Like stratified sampling, the population is divided into groups (or *clusters*), and the groups are randomly selected for inclusion in the sample. All elements in the population must be assigned to a cluster, and no element can be in more than one cluster. From these groups, the metrics of interest are measured. Cluster sampling differs from stratified sampling in that only the clusters that have been randomly selected are used for measurement. In addition, the entire cluster is measured, whereas in stratified sampling, each stratum is sampled. Clusters should generally be heterogeneous in nature. Given a fixed cluster size, cluster sampling may be less precise than stratified sampling or simple random sampling, but it incurs a smaller cost on each individual sampling unit. Consequently, the loss of precision can be diminished by increasing the overall sample size (i.e., the number of clusters times the cluster size).

In the context of processor simulation, a cluster is a contiguous group of cycles from the simulation [15]. Because the behavior of such a cluster is hard to measure without full simulation, instructions from the dynamic instruction stream are often used as a proxy for cycles. Each of the randomly chosen clusters is then simulated in order to estimate any attribute desired by the user (e.g. IPC, cache performance, speedup, etc.). The larger the sample, the more likely the estimates obtained from that sample will be correct. Very large samples will be extremely accurate, but at the cost of long simulation times. Small samples will be simulated very rapidly, but may yield inaccurate estimates. Therefore, care must be taken to select an appropriate *sampling regimen* [15],[16]. The sampling regimen defines the number of clusters and the cluster size for a particular workload.

In multiprocessor simulations, the approximation that cycles equal instructions is highly inaccurate. Figure 3 helps to explain why. The figure details the per-thread warming period, cluster/detailed execution, and functional simulation and warmup phases for a multiprocessor executing a single multithreaded application. The number of cycles needed to execute a sequence of instructions is dependent on the degree of parallelism, and thus a complex function of the workload. Finding an appropriate substitute proxy for cycles is an open research topic.

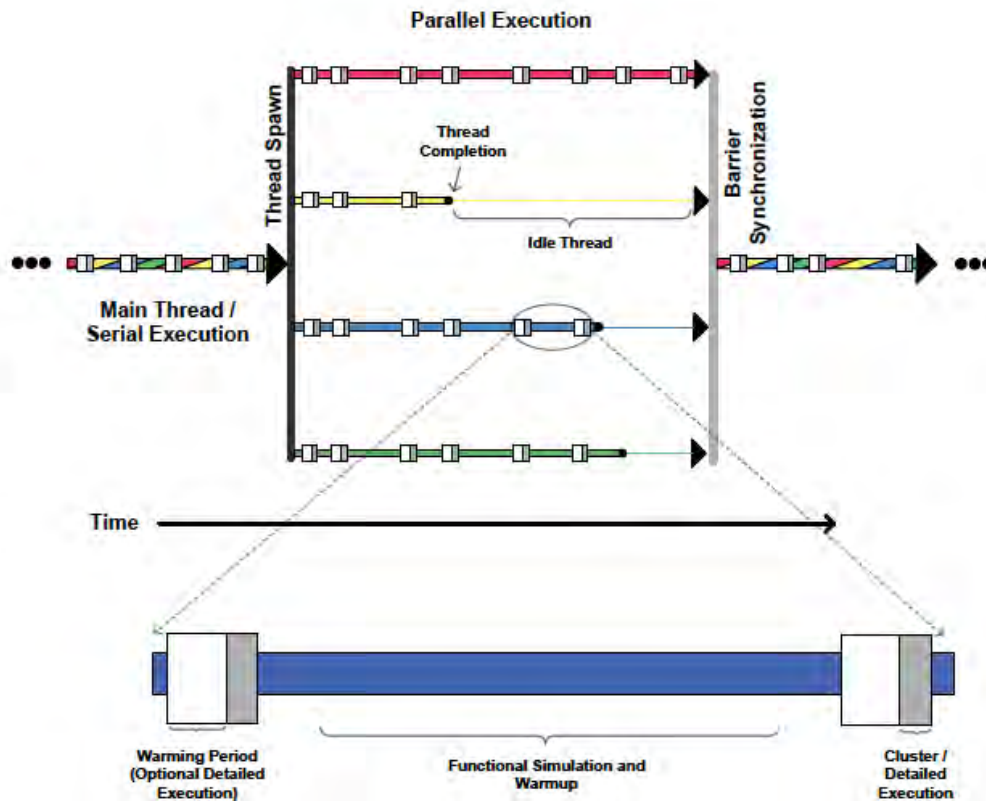


Figure 3: Cluster sampling for multiprocessor architectures.

As with processor simulation, it is still desirable for cluster locations to be selected randomly from executing threads throughout the program. In between clusters, new functional warming algorithms combined with detailed warming are required to recover state that is lost while skipping cycles. The region between cluster measurements is referred to as the *skip-region*, or simply the *gap*. *Functional warming* uses data obtained during the functional emulation of the workload in order to approximate the system state. During this phase of execution, events are immediately processed and timing information is omitted. (For example, reference streams operating on caches are applied immediately without simulating those latencies such operations would incur.) Techniques used in the functional warming of simulator state are called *warmup methods*. An optional detailed warming period follows functional warming and can be used to reconstruct state in a timing-specific manner. Detailed warming simulation and cluster simulation are similar, except the warming region doesn't influence collected statistics in the detailed warming simulation, whereas it does in cluster simulation. Either reduces nonsampling bias encountered during detailed warming from significantly affecting sample accuracy.

Utilizing these three phases of execution, the workload will transition between regions of serial and parallel execution. During serial execution, single-core cluster sampling techniques can be

applied to warm system state because the interleaving of thread events and timing behaviors do not exist. Once threads are spawned, cluster phases are randomly collected across all system threads simultaneously. In other words, the location of the cluster is random, but all clusters occur temporally at the same time. Since threads may progress at different rates, functional warming should approximate their relative progress before the next detailed warming and cluster quantum are measured.

Several functional warming techniques have been studied to efficiently recreate simulator state (originally for cache simulation [12],[13],[17],[18],[19] and then later extended to processor simulation [14]). Since detailed execution occurs on only a small fraction of the entire program, simulation times are now bounded by skip-region processing between clusters. The most accurate warmup method is SMARTS [20], because all data in the skip-region are functionally applied. Consumption of all skip-region data provides an accurate representation of system state, but is also heavy-handed and expensive in terms of simulation effort. Due to locality, the functional execution of many instructions could be omitted with minimal effects on sampling accuracy. Warmup methods may exploit system characteristics to infer state properties [11],[12],[13],[21] in order to approximate the functional state produced through SMARTS-style functional simulation.

Another popular technique used to accelerate simulation is SimPoint [10],[22]. SimPoint uses data obtained from a fast functional simulation of the program to classify program behavior. Using BBV (Basic Block Vector) information, clustering techniques are applied to condense the workload behavior into a small number of representative data points. SimPoint has been successfully used to accurately estimate entire workload behaviors, and can be thought of as a type of sampling. However, due to the systematic sampling of events from the program, program periodicity may cause sampling bias effects to skew measured results. Although extending SimPoint to multiprocessors is an interesting research direction, it has not yet been studied.

2.5 Summary: Predictable Design

The approach advocated here for predictable design is based primarily on simulation of the desired processor. The simulation is accelerated using statistical techniques that allow the error in the results to be predicted. Moreover, in terms of choosing between architectural approaches, predictable design also advocates choosing architectural approaches that allow for accurate performance estimations with less reliance on detailed simulation, for instance, using message passing paradigms instead of shared memory paradigms. By reducing the cost of simulating a given architecture, predictable design methodologies will allow EES systems to be optimized for high performance subject to very tight design constraints through affordable exploration of many alternatives in the design space.

SECTION 3

APPLICATION ANALYSIS AND BENCHMARKS FOR EMBEDDED EXTREME SCALE SYSTEMS

3.1 Introduction

The space of embedded high performance applications includes several computing styles, including at least streaming applications, graph applications, unstructured knowledge extraction, and combinations of these into an overall multi-stage processing flow. Traditionally, the greatest emphasis in embedded HPC for DoD platforms has been streaming applications, often associated with radar, sonar, EO/IR, and similar sensors. Figure 4 illustrates the general structure of an embedded streaming sensor signal processing system. This figure implies one processing system shared by several sensors. This may be the case for some scenarios, while in others each sensor may have its own dedicated processing system. In still other cases, there may be a hybrid structure, where each system has some amount of front-end processing, while the back-end processing and knowledge generation occurs in a shared processing system.

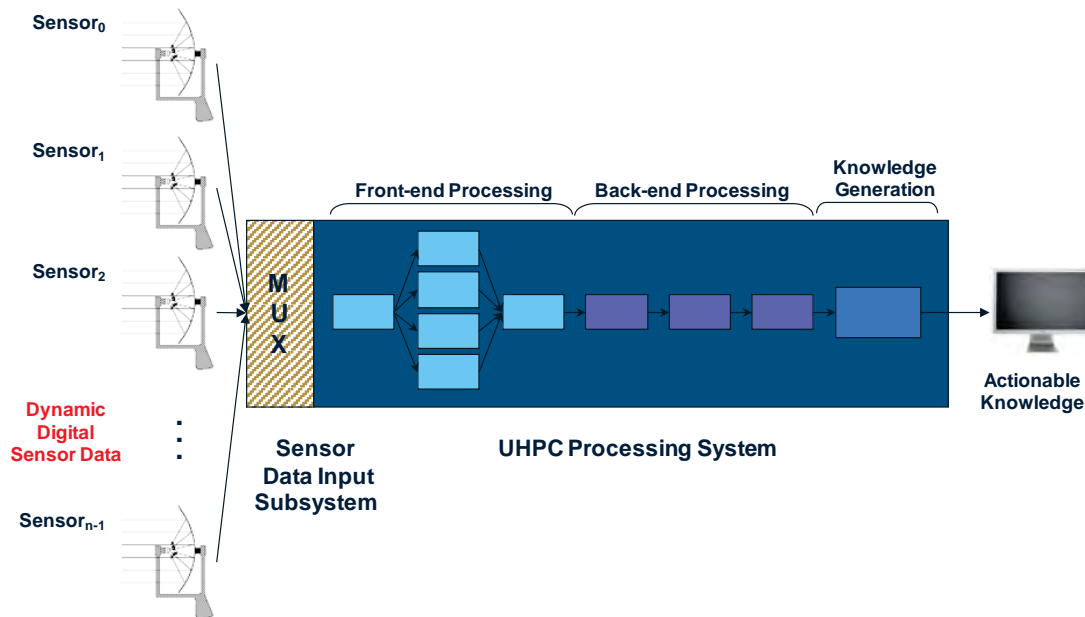


Figure 4. General structure of an embedded streaming sensor signal processing system.

Streaming sensor computing is primarily characterized by high throughput numeric processing of a flow of input data from external sensors. Typically, this flow consists of discrete blocks of data of a fixed volume, arriving at a fixed cadence. Representative sensor types include multichannel radar and electro-optic, infrared, and video imagers. Typical application domains include multi-modal image formation, surface moving target indication, computer vision for robotics and autonomous vehicles, surveillance, and communications. Streaming sensor

processing is frequently used in interactive systems or as a component of feedback control systems that also impose a latency requirement.

Challenging streaming applications present large volumes of data generated by the sensors, large volumes of intermediate data, relatively low arithmetic intensity (math operations per data word read or written), large volumes of computations repetitively applied to advancing frames of data, predictable data access patterns, relatively low portions of data flow that constitute feedback, and relatively small numbers of dynamic branches within the calculations. The latency requirement is often equivalent to many input block periods, making pipelined task parallelism an effective acceleration approach.

Traditional streaming sensor data is captured as fixed-point numbers at bit depths of 16 or less and organized into two- or three-dimensional data arrays representing a discrete frame of data. The data may pass through front-end processing performed with fixed-function or reconfigurable hardware to transform their domain, reduce their size, and convert them to the appropriate numerical format (which is usually single- or double-precision floating point) for further processing. Subsequent processing typically consists of fast Fourier transforms (FFTs) and related transforms, n -dimensional (n -d) convolutions and correlations, n -d covariance estimations, linear solvers for structured systems, thresholding, low-order spatial averaging, and similar localized signal processing and linear algebra operations. These kernels differ widely in their spatial and temporal locality characteristics, as well as their opportunities for fine-grain parallelism. Corner turn operations to optimize access patterns are common. The final output of this processing chain is a relatively small number of results, such as potential objects of interest for a processor imaging system or language tokens for a speech processing system.

The emergence of exascale technology will enable more complex and challenging streaming sensor applications to be deployed. The availability of large numbers of floating point operations per time enables more complex operations on larger data sets. Examples of these operations are full-rank space-time adaptive processing systems and thin-plate-spline modeling for frame-to-frame registration of high resolution images. Such operations increase the arithmetic load of the algorithms and can also cause intermediate data set sizes to increase through the processing chain. These changes enable more demanding applications domains such as persistent, wide-area surveillance with knowledge extraction. Future streaming sensor applications will therefore have increased requirements in arithmetic capability, external I/O capabilities, internal memory capacity, and bandwidth. Future streaming sensor processing may add new layers of processing, such as self-aware, self-tuning capabilities to improve processor efficiency, load balancing, and resiliency.

3.2 Computational Requirements for Examples of Embedded Extreme-Scale Computing Applications

3.2.1 KAPE

STAP (space-time adaptive processing) relies on an estimate of the statistics of the interference components (clutter, jamming, electromagnetic interference [EMI]) of the radar data to extract signals of interest. These statistics must represent the radar environment when it is free of targets. Here the focus on the clutter statistics. The traditional approach is to test for a target at a

particular range while estimating the clutter statistics from other nearby ranges. This approach assumes that the interference statistics don't vary as a function of range, so that averaging over nearby ranges can provide a good estimate of the true statistics at the range of interest. In many cases this assumption is not valid, for example in urban settings where the terrain characteristics change over short distances. The object of KAPE [23] (Knowledge-Aided Parameter Estimation) is to estimate the clutter statistics *without* training on nearby ranges. Instead, prior knowledge of the scene is used to construct the covariance matrix needed for the adaptive processing.

Conventional STAP computes a covariance matrix by averaging the outer-product of data vectors for a number of neighboring range bins. KAPE computes an initial "bootstrap" guess at this covariance matrix and then tests it against the measured data. This procedure involves perturbing the covariance matrix parameters and testing for the "best fit" with the measured radar data. This testing procedure primarily involves solving systems of linear equations.

KAPE is a GTRI algorithm developed in the mid-2000's and derived from research originated under DARPA's KASSPER (Knowledge-Aided Sensor Signal Processing and Expert Reasoning) program. Figure 5 illustrates the high-level functional flow of the algorithm. KAPE achieves very good detection performance compared to conventional STAP, as shown in Figure 6, but has not been widely adopted due to high compute requirements. However, these same requirements are a plus for benchmarking extreme scale computing systems. Example configurations require approximately 200 Gflops, but more challenging modes are easily defined. KAPE operates on a typical radar datacube in the slow-time and phase center dimensions, which implies that the whole datacube must be buffered.

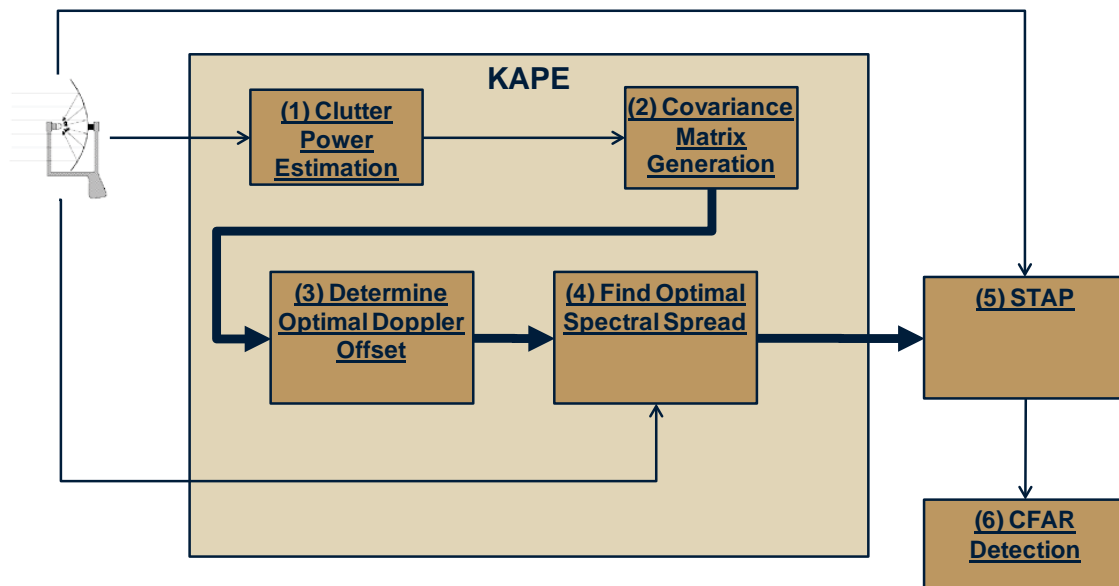


Figure 5. Functional overview of KAPE detection processor.

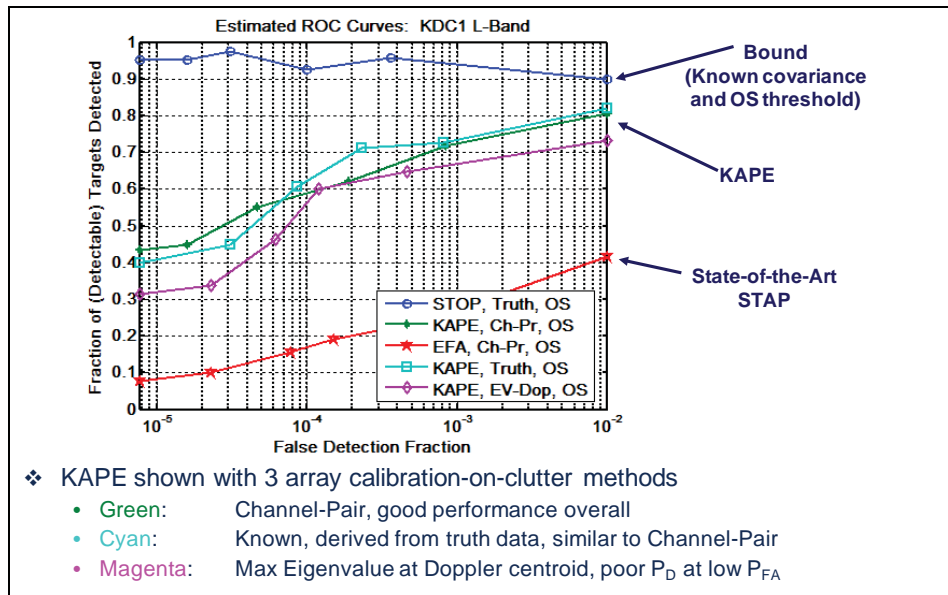


Figure 6. Example of KAPE detection performance improvement. From [23].

Table 1 lists system parameters appropriate for various applications of the KAPE technology. The various scenario options are as follows:

- *KAPE Example Value*: this is the parameter value used in the original KAPE analysis
- *WAS*: Wide Area Surveillance, “traditional” GMTI as employed by large sensor platforms such as JSTARS and Global Hawk. The system is required to periodically (say, every 10 seconds) search for ground movers over a larger region (perhaps several thousand square kilometers). The radar accomplishes this task by electronically steering a TX and RX beam over the scene. The dwell time on each step is relatively short (many 10s of ms, perhaps on the order of 0.1 s), long enough to generate sufficient signal-to-noise ratio (SNR) and Doppler resolution on large ground movers. Range resolutions are moderate: fine enough to match the size of large ground vehicles, but not so fine to over-resolve these vehicles, or to generate an excessive number of range bins across the ground swath. Three channels are sufficient to cancel clutter and measure the angle to large targets.
- *UAV Dismount*: As compared to WAS, a smaller radar uses a longer, concentrated dwell to detect people moving about a relatively localized area of interest. Dwell time is much longer than WAS to cull out dismount Doppler signatures. Bandwidth is higher to achieve better range resolution and thus better isolate a dismount in range, but the overall range swath size is much less than WAS, and the number of range bins far fewer. Four channels improve STAP clutter cancellation and angle estimation to small, slow targets.

- *Persistent Dismount*: The platform orbits an area of interest and continuously spotlights the ground with a (perhaps spoiled) transmit beam. The dwell is continuous. As compared to UAV Dismount, the PRF is higher and more channels are available to ensure sufficient along-track sampling intervals. The area surveyed is much larger than the UAV Dismount case (several km versus hundreds of meters), so there are more range bins and Doppler bins. Otherwise this case is similar to UAV Dismount. An example of this concept is the system discussed further in Section 3.2.2.
- *Challenge Scenario*: A notional scenario scaled to require a computational capability of approximately one-half exaflops (Eflops). It is similar in many ways to the UAV Dismount scenario, but with an order of magnitude increase in the number of range bins, due either to finer resolution, larger swath, or both.

Table 1. KAPE System Parameters for Various Operational Modes

Variable	Description	KAPE Example Value	Challenge Scenario	WAS	UAV Dismount	Persistent Dismount
N_{sp}	Number of (spatial) channels	6	12	3	4	6
N_p	Number of pulses	32	1024	128	1,024	2,048
L_r	Number of range bins	200	50,000	60,000	4,000	15,000
N_{az1}	Number of azimuth angle samples	38	1024	128	1024	2048
N_{os}	Oversampling Factor: $N_{os} = N_{az2} / N_{az1}$	5	5	5	5	5
N_{fo}	Number of Doppler offset frequencies to test.	13	13	13	13	13
N_w	Billingsley wind speed value	31	31	31	31	31
N_{pad}	Length of temporal steering vectors	129	2048	256	2048	4096
L_{win_r}	Number of training bins for the CFAR detector	10	10	10	10	10
t	Time between collections	1/10	1	1/8	2	1
-	Bandwidth (MHz) / Range Resolution (m)	-	-	60 / 3	180 / 1	600 / 0.3
-	Range Swath Depth (km)	-	-	150	3	5
-	PRF (Hz)	-	-	1,000	500	2,000

Table 2 lists the estimated computational requirements in flops for each of these KAPE scenarios. The loads range from 167 Gflops for the “example” case to 0.572 Eflops for the “challenge” case. This table also indicates the class of computing technology required to provide these levels of performance and the estimated power requirements as of late 2010. At the low end, the compute requirements of the “example” scenario could be hosted on a mobile graphical processing unit (GPU) such as the NVIDIA GT525M requiring about 23 W. The WAS scenario, at 455 teraflops (Tflops), would utilize a little more than one-fifth the capacity of the Tianhe-1A supercomputer, the fastest machine in the world in November 2010 [24], but at a power cost of approximately 716 kW. The UAV dismount would require almost three times the capacity of the Tianhe-1A and about 16 MW of power, while the persistent dismount and challenge scenarios are essentially unrealizable using 2010 technology. The WAS and UAV dismount scenarios, while possible, require data center-like fixed installations; they are not realizable as field deployable systems, let alone embedded systems.

The row labeled “UHPC” estimates the system type and power requirements to implement these same capabilities, assuming that the speed and power goals of DARPA’s Ubiquitous High Performance Computing program (UHPC) [25] are met. The WAS system would require one-half of a standard rack and approximately 10 kW, easily achievable in field-deployable systems. The UAV dismount would require a larger installation, but still one that could be established at remote bases. The persistent dismount and challenge scenarios, while implementable with UHPC technology, would still require large data centers.

Table 2. KAPE Systems Computational Summary.

Block	Example	WAS	UAV Dismount	Persistent Dismount	Challenge
Power Estimation	11.7e+06	23.6e+09	134e+09	3.02e+12	5.03e+12
Covariance Generation	11.3e+09	45.4e+12	2.75e+15	186e+15	309e+15
Doppler Offset	28.5e+06	189e+06	188e+09	4.99e+12	4.99e+12
Spectral Spread	629e+06	10.5e+09	5.70e+12	154e+12	154e+12
STAP	4.80e+09	11.4e+12	871e+12	83.5e+15	263e+16
CFAR	4.70e+06	2.80e+09	1.49e+09	11.2e+09	18.6e+09
Total	16.7e+09	56.9e+12	3.63e+15	269e+15	572e+15
Latency (s)	0.1	0.125	0.5	2	1
FLOP/s	167e+09	455e+12	7.25e+15	135e+15	572e+15
May 2011:	NVIDIA GT525M Mobile GPU (23W)	1/5 Tianhe-1A (716 kW)	3× Tianhe-1A (16 MW)	No	No
UHPC	Sub-proc (≈2 W)	Half-rack (≈10 kW)	1’s of racks (≈145 kW)	Data Center (≈2.5 MW)	National Asset (≈11 MW)

The KAPE algorithm and Table 2 demonstrate the need for advanced computing technology. KAPE is an advanced algorithm that significantly improves the ability to detect ground movers from surveillance platforms, enabling advanced detection, tracking, and targeting capabilities such as the UAV and persistent dismount tactics. However, this capability is not deployable in realistic systems without the major improvements in computational speed and power efficiency envisioned by the UHPC program.

3.2.2 Persistent Wide-Area Radar Surveillance

There is significant interest in the DoD in developing systems for persistent wide-area surveillance (WAS) from airborne platforms using video, electro-optic/infrared (EO/IR), and synthetic aperture radar (SAR) sensors singly or in various combinations. SAR is of particular interest because of its weather penetration ability, giving a 24/7 WAS capability. An airborne SAR-based persistent WAS can provide a series of very fine resolution radar images of a region of interest (ROI). The SAR image series enables a “video SAR” capability that in turn supports identification and tracking of ground moving targets over significant time periods [26],[27].

Figure 7 illustrates the top-level signal processing flow for a SAR-based persistent WAS system. The optional digital spotlighting step permits formation of the image of just a subregion of the collection area. Multiple subimages can be formed from one data set. Spotlighting involves a number of 1D FFTs, along with element-wise vector multiplies and downsampling, all on complex-valued data, as shown in Figure 8. The image formation step is a set of 1D FFTs followed by a general backprojection algorithm. Backprojection [28], shown in Figure 9, is simply a triply-nested loop, but therefore has an $O(N^3)$ computational complexity, creating a major computational load that also scales rapidly with the problem size. Once formed, image data is compressed for transmittal to a ground station for processing. For this analysis, JPEG2000 was taken as a representative lossy compression method. However, the computational load for image compression is much less than that for image formation, and so is not discussed further here.

GT has estimated the image formation computational load of a basic SAR-based persistent WAS system. Full details are not presented here, because they are superseded by the more aggressive Streaming Sensor Benchmark analysis presented in Section 3.2.3. However, to get a rough sense of the problem, estimated computational loads for the basic image formation only (no spotlighting) are given in Table 5. The 1,024×1,024 case corresponds roughly to a small ROI on the order of 0.3 km on a side; the 4,096×4,096 to an ROI on the order of 1.25 km on a side; and the 66,666×66,666 to a future, much larger ROI size. The small image, at 43.8 Gflops, represents a nontrivial computational load, but one that is well within current capabilities, even for medium-scale embedded systems. The medium-scale ROI, at 2.8 Tflops, probably exceeds current embeddable computing capability (though not likely for much longer), but is easily achieved in a ground station. However, at 12.1 Pflops, the large ROI considerably exceeds the 2.57 Pflops capability of the Chinese Tianhe-1A supercomputer.

Table 3. Computational load for SAR-based persistent WAS image formation.

	Image Size (pixels)		
	1,024×1,024	4,096×4,096	66,666×66,666
Estimated FLOPS (no spotlighting)	43.8×10^9 (43.8 Gflops)	2.8×10^{12} (2.8 Tflops)	12.1×10^{15} (12.1 Pflops)



Figure 7. Top-level SAR-based persistent WAS processing flow.

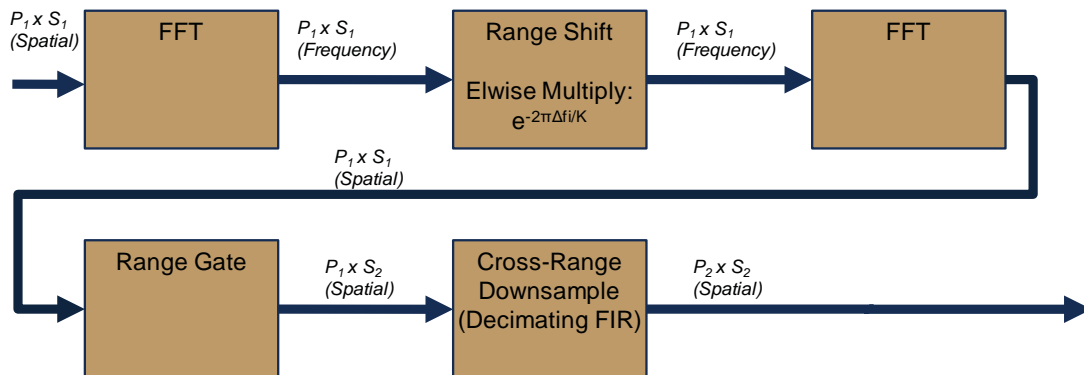


Figure 8. Processing flow for optional digital spotlighting step.

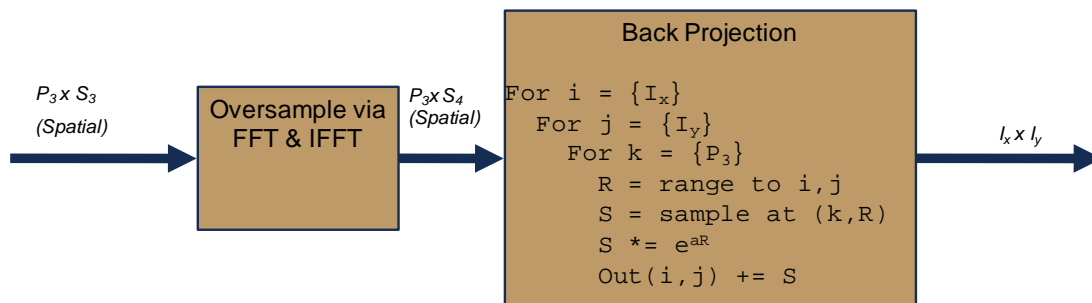


Figure 9. Backprojection algorithm.

The use of tiling and digital spotlighting can mitigate these loads somewhat. Increasing the number of tiles (therefore making each tile smaller) increases the total spotlighting computational load, but reduces the load in the image formation step. These opposing trends result in a distinct minimum for the total load at a particular tiling factor that depends on details of the problem definition. One calculation for the large-ROI case estimated the minimum to occur at a tiling factor of 128, where total estimated flops count was reduced from 12.1 Pflops to 0.14 Pflops.

The SAR-based persistent WAS scenario was selected as the basis for initial definition of a Streaming Sensor Challenge Problem, which is the subject of the next subsection.

3.2.3 Streaming Sensor Challenge Problem Development

Based on the experience gained in analysis of the foregoing applications, GT developed an initial outline of a “streaming sensor challenge problem” (SSCP). The purpose of the SSCP is to provide a publicly releasable computing problem specification that exhibits types and quantities of computation that are generally representative of DoD-relevant streaming sensor problems such as those discussed above. A good challenge problem (CP) will exhibit at least the following characteristics:

- ◆ Close correlation to relevant DoD missions;
- ◆ Public releasability of specification and associated data;
- ◆ Positive correlation between improved benchmark results and improved mission value
- ◆ Ability to evaluate expected system axes of variation;
- ◆ Communication, memory, and computation loads large enough to stress current and future computing systems; and
- ◆ Sufficiently abstract expression to support innovation at all levels of the hardware and software system stacks.

The SSCP would be comprised of a written problem specification, possibly containing several variants based on different parameterizations to provide different computing loads; an “executable specification” in one or more languages, such as MATLAB or C; and one of more data sets to drive the application.

The version of the SSCP described here is serving as the starting point for the ongoing development of the UHPC program challenge problem #1. The UHPC challenge problem will further refine the SSCP and may expand it to add a video imaging sensor and multi-sensor fusion.

3.2.3.1 Overview

The Streaming Sensor Challenge Problem models the computational demands of streaming sensor computing tasks, which transform large volumes of raw sensor data to a smaller volume of actionable knowledge tokens. The transformations generally consist of highly regular, repetitive application of static sequences of calculations on fixed-size data blocks that arrive for processing at a constant rate in time. Challenging problems consist of very large volumes of data input, several large multidimensional calculations, large intermediate data sets, and very high internal bandwidth requirements.

This type of processing is present in many DoD-relevant missions, including but not limited to multi-modal image formation using radio frequency (RF) and EO/IR sensors, surface moving target indication, computer vision for robotics and autonomous vehicles, surveillance, and communications. Increasing numbers, quality, and interconnectedness of global sensors have elevated the required computing capabilities to levels consistent with extreme scale computing goals.

3.2.3.2 Challenge Problem Description

The proposed SSCP models a wide-area, high resolution radar persistent surveillance mission similar to that discussed in the previous section, but with additional “knowledge formation” steps applied to the SAR images to generate detections as shown in Figure 10.

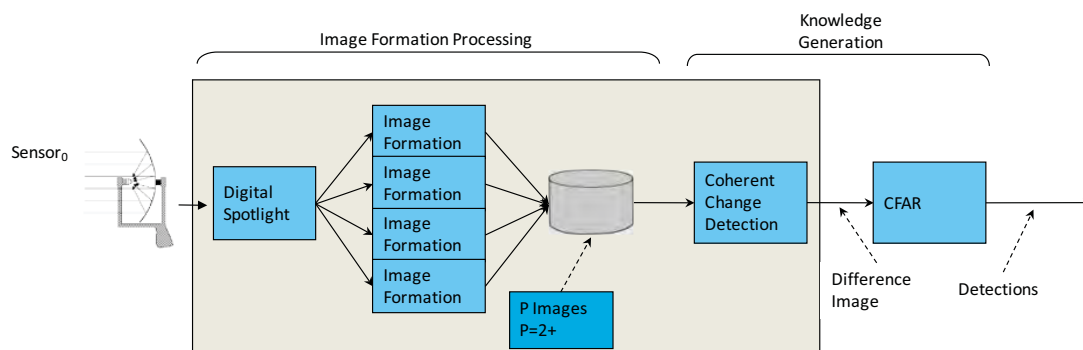


Figure 10. Top-level SSCP processing flow.

In the image formation portion of the mission, an airborne radar system flies a repeated path around a target area to be observed. The system illuminates the target area with radar pulses, capturing the complex reflected signal voltage from each pulse at a number of points in time. These returns, along with the time of each pulse and the position of the transmitter and receiver at each pulse form the main inputs to the CP. An optional digital spotlighting step creates tiled subsets of the full target area, reducing the overall scaling order of the computational requirements for large images. The returns are backprojected to form a synthetic aperture radar image of a specific resolution and size. Overlapping sets of pulses are used to form successive images at a specified cadence.

In the knowledge formation portion of the mission, successive images are registered via a two stage process comprising a global affine transformation and a thin plate spline warping. The first operation accounts for any bulk relative shift between the two images, while the second accounts for local distortions. Coherent change detection (CCD) is then applied to between pairs of registered images to identify pixel locations of significant change, which form the output tokens for the challenge problem.

The CP includes a set of computational kernels that vary in the relative load of computation and communication, with regular, but diverse data access patterns. These kernels will stress the ability of UHPC systems to perform dense, very high speed spectral, signal processing, and linear algebra computations with large datasets and diverse data access patterns.

- ◆ Digital Spotlight: A series of 1D FFTs along the samples of each pulse are executed, along with a pair of element-wise multiplies, an array cull in one dimension, and a FIR reduction filter.
- ◆ Backprojection: For each pixel location in the formed image, a corresponding range to the receiver on each pulse is computed. The return from that range is estimated by linear interpolation from the adjacent return samples for the given pulse. The pattern of returns is correlated with the point spread response of an ideal radar reflector at the pixel location to form the image output value.
- ◆ Image Registration: Affine [29]: For each of a specified number of control points in the image, a best-fit offset vector is found for a neighborhood of a specified size relative to the reference image within a specified radius. A matrix with six columns and a row for each control point is formed and a least squares solver is used to form an affine transformation that is then applied to each location in the output to find the sample location in the formed image, which is bilinearly sampled to form the registered image. This flow is illustrated in Figure 11.
- ◆ Image Registration: Thin Spline Warping [30]: For each of a specified number of control points, a best-fit offset vector is found, as above. These vectors form a two column matrix which is left multiplied by a square control point weighting matrix to form a two column warp weight matrix. For each output location, an offset is formed by a weighted sum of warp weight rows. This offset is used to find the location for a bilinear sample of the affine-registered image to form the final image. This flow is illustrated in Figure 12.
- ◆ Coherent Change Detection [31]: Each pixel in the image is assigned a coherence score by correlating a neighborhood around the pixel with that same neighborhood in the

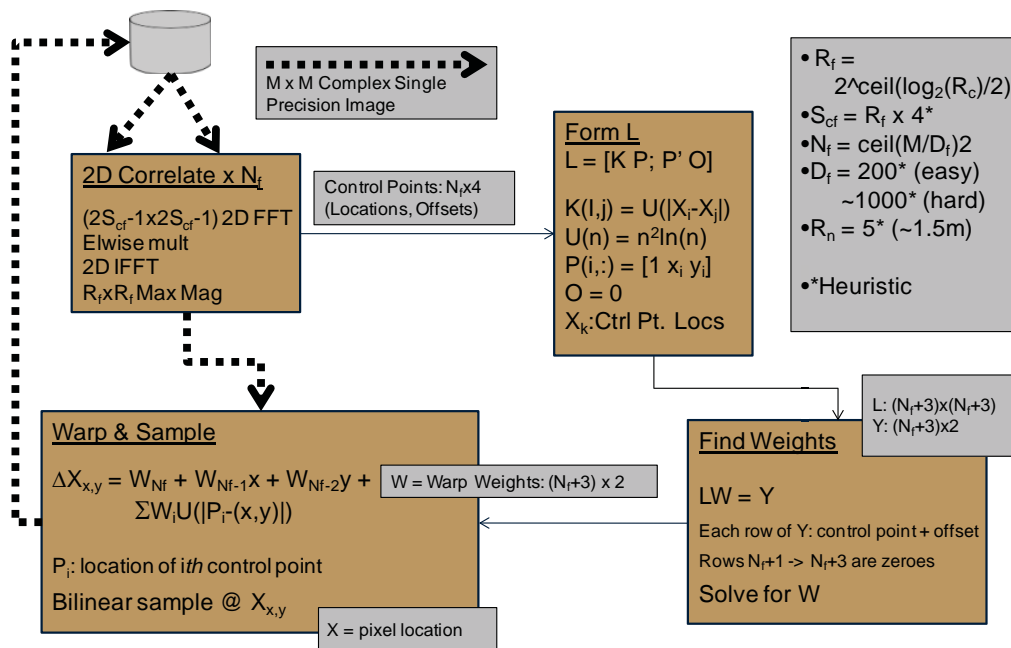


Figure 12. Thin spline registration processing flow.

Table 4. Tentative Challenge Problem Scenarios.

Scenario	1	2	3	4
Ground Area (square edge size, m)	609.6	1219	2438	9753
Image Size (edge size, pixels)	4000	8000	16000	64000
Pulses per Image	4800	9600	19200	76800
Samples per Pulse	4000	8000	16000	64000
Pulses per Second	1084			
Image cadence (images per second)	1			
Affine registration control points	3629	14,513	58050	928799
Thin-spline registration control points	1452	5805	22320	371520
CCD neighborhood size	5×5			
CFAR neighborhood size	15×15			

Table 5. Scenario Loadings.

Scenario		1	2	3	4
Floating Point Operations	Digital Spotlight	90.1×10^9	1.47×10^{12}	6.27×10^{12}	442×10^{12}
	Backprojection	326×10^9	1.31×10^{12}	10.4×10^{12}	334×10^{12}
	Affine Registration	557×10^6	5.04×10^9	20.2×10^9	322×10^9
	Thin-spline registration	20.0×10^9	8.02×10^{12}	147×10^{12}	184×10^{15}
	Coherent Change Detection	2.36×10^9	400×10^9	1.6×10^{12}	25.6×10^{12}
	CFAR Detection	8.36×10^6	1.44×10^9	5.76×10^9	92.1×10^9
Total flops		440×10^9	11.2×10^{12}	166×10^{12}	185×10^{15}
Input bandwidth (bps)		444×10^6	556×10^6	1.33×10^9	5.33×10^9

Table 6 describes the general computational style of each of the major functional stages of the SSCP. This challenge problem depends primarily on three classes of computing: 1D FFTs; solution of large systems of linear equations; and a triple-nested backprojection loop. While the CFAR processing introduces sorting to the mix, it is a small fraction of the overall processing.

Table 6. Computational Style of Major Functions.

Section	Dominant Compute Styles
Digital Spotlight	A few large FFTs, array multiply
Backprojection	Embarrassingly Parallel, some read coherence
Affine Registration	Many small FFTs, Large system solve, large matrix multiply
Thin-spline registration	Many small FFTs, Large system solve, large matrix multiply
Coherent Change Detection	Many small FFTs
CFAR Detection	Many small sorts

3.2.4 Autonomous Vehicles

A very different class of embedded computing with potentially extreme processing requirements is that of autonomous vehicles. In 2007, DARPA conducted the "Urban Challenge", an event that sought to advance the technology for building an autonomous vehicle capable of driving in traffic, performing complex maneuvers such as merging, passing, parking and negotiating intersections [32]. The event was the first time autonomous vehicles interacted with both manned and unmanned vehicle traffic in an urban-like environment. Georgia Tech's "Sting Racing" team participated in the Urban Challenge. The Sting robot vehicle, a retrofitted Porsche Cayenne, is shown in Figure 13. The radar is used to detect obstacles for avoiding head-on collisions. The laser range finders provide terrain mapping. Six video cameras detect other

vehicles or traffic at a four-way stop. The GPS receiver on the top of the robot is coupled to an IMU inside the vehicle for navigation. The computational hardware architecture is illustrated in Figure 14, while the software architecture is diagrammed in Figure 15. The LOIS and stopline algorithms comprise the *perception module*, while the mapper, planner, and controller comprise the *planning module*. Note that, unlike the streaming signal processing of the KAPE, WAS, and SSCP applications, Sting emphasizes computer vision and robotics algorithms: image analysis and understanding; mapping, route planning, and obstacle avoidance; and robot control.

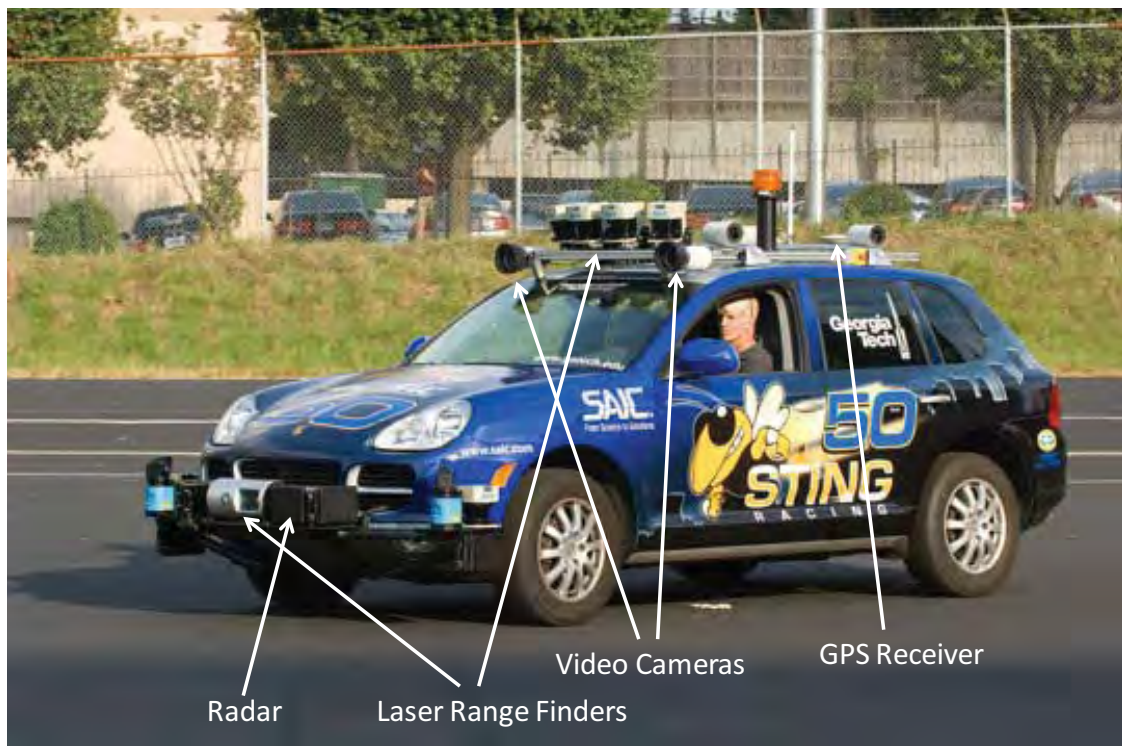


Figure 13. GT's Sting autonomous vehicle.

3.2.4.1 Sting Computational Analysis

The entire Sting software suite ran on gigabit network of 8 Dual-Core Intel XEON 5120 processor-based computers [33], so clearly this is not an extreme scale realization. Nonetheless, GT analyzed the computational requirements to understand the types of processing involved and the current (2007) computational loads. GT then drew from the robotics literature to posit a more advanced system that could be scaled to petascale and higher computational requirements and would achieve superior performance [34].

Consider the perception module first. The LOIS module is a lane detection algorithm. It is paired with a stopline detection algorithm to analyze the primary video camera inputs to

determine land boundaries and stoplines at intersections. These detections provide basic constraints to the mapping, planning, and control modules.

As seen in Figure 13, two forward-looking video cameras are mounted just above the car's windshield. The STING cameras were Prosilica GC650 models that provide color and black & white output at up to 90 frames per second (fps). These frame rates are high; typical camera frame rates are 30 fps.

Figure 16 illustrates the major steps in the LOIS lane tracking algorithm. GT estimated the computational load associated with each step, considering both high-and low-resolution camera modes of 592×800 and 296×400 pixels, respectively; both were full-color images. While the camera provides data at up to 90 fps, image analysis algorithms are often applied at lower rates, e.g. 10 or 20 fps, and sometimes as little as 1 fps. The lower analysis rate may be chosen because it is adequate to achieve the desired results, or because of computational limitations. GT used a relatively high value of 20 fps for estimating the computational load in order to support relatively high quality lane detection.

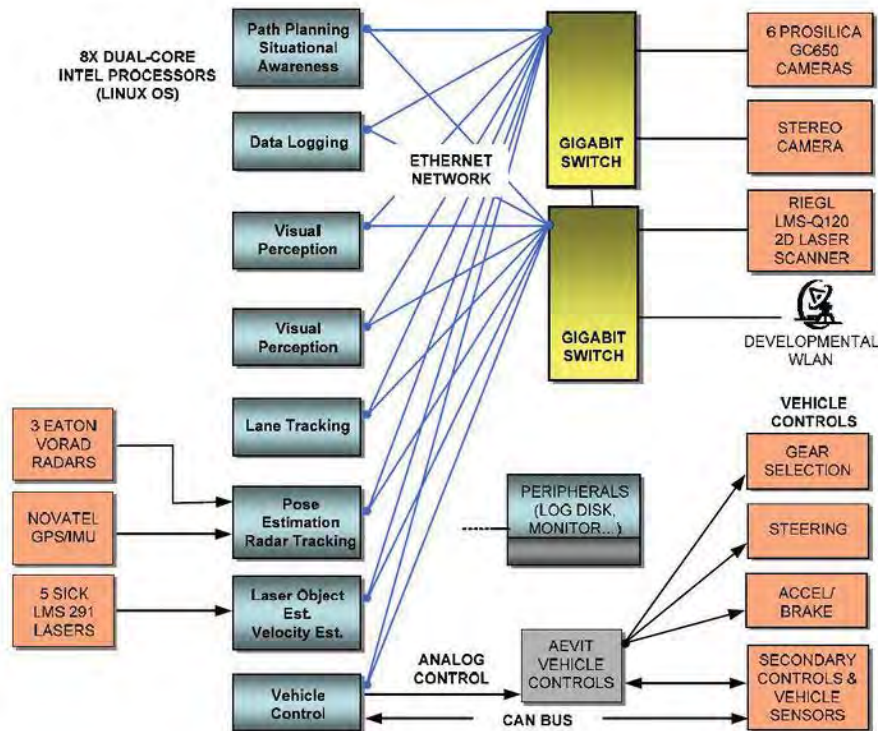


Figure 14. Sting hardware architecture.

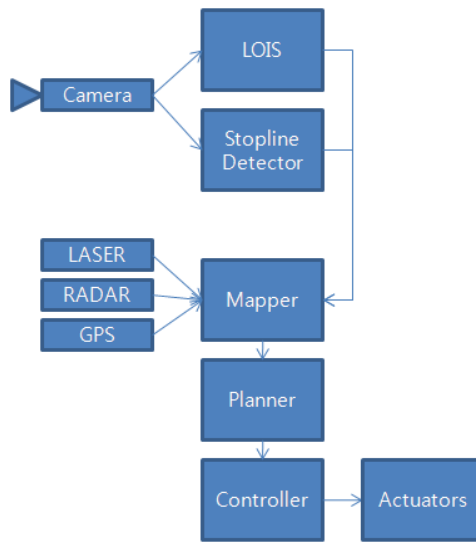


Figure 15. Sting software architecture.

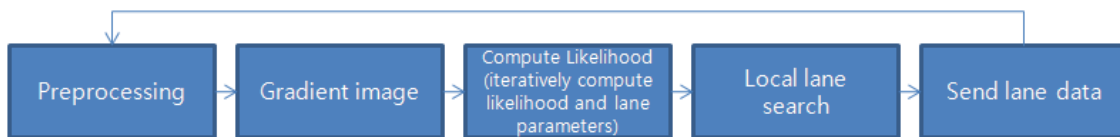


Figure 16. Major steps in the LOIS lane tracking algorithm.

The key result is highlighted in Table 7. At 20 fps, lane detection requires about 172 million operations per second (Mops) at low resolution, and about 355 Mops at high resolution.²

Table 7. Estimated computational requirements of the LOIS lane tracking algorithms.

Operation	Computational Load (Mops)	
	High Resolution	Low Resolution
Preprocessing	56.8	14.2
Gradient Image	75.6	18.9
Likelihood	223.0	138.9
Total	355.4	172.0

² Because the processing is a mixture of integer and floating point operations, the Mops metric is used instead of Mflops.

Figure 17 illustrates the major steps in the stopline algorithm, while Table 8 estimates the computational load at about 192 Mops for the high resolution case, and 36.5 Mops for the low resolution case.

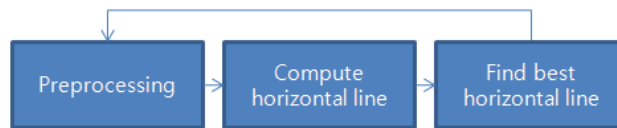


Figure 17. Major steps in the stopline algorithm.

Table 8. Estimated computational requirements of the stopline algorithm.

Operation	Computational Load (Mops)	
	High Resolution	Low Resolution
Preprocessing	74.5	18.6
Hough Transform	109.0	13.6
Select Optimum Line	8.4	4.2
Total	191.9	36.5

Turning now to the planning module, the major steps in the mapper, planner, and controller are each shown in Figure 18 (a), (b) and (c), respectively. The mapper creates a grid-based map and a dynamic graph-based map from the sensor returns such as radar and laser range finder data. In the Sting system, the mapper also handles obstacles and lanes. The planner is the A* (pronounced "A star") algorithm, a best-first graph search algorithm that finds the least-cost path from a given initial node to one goal node (out of one or more possible goals). The controller controls the Sting car functions such as stop/start, speed, setting turn signals, and so forth.

The computational load of these steps is highly variable based on a number of scene-dependent parameters such as the number of laser range finder detections, the number of detected curb points, the number of obstacles being tracked, and so forth. Based on reasonable assumptions in an Urban Challenge environment, and again assuming a 20 frames second frame rate, GT estimated the computational load for the mapper to be on the order of 1.5 Gops. The computational loads for the controller are considered negligible in comparison. The computational load for the planner is also considered low, although the central component of the A* algorithm, which involves a graph search, is of $O(B^D)$ complexity, where B is the branching factor and D is the depth of the tree from the robot to the goal. This load could grow very rapidly with deep trees and large branching factors.

Combining this result with those from the perception module gives a total estimated computational load of roughly 2 Gops in the high-resolution case. Thus, while the Sting architecture is useful for illuminating the types of functions needed for an autonomous road vehicle, the 2007 implementation does not pose an extreme scale computing challenge. For this reason, GT next postulated a more advanced algorithm suite, drawing on the robotics literature for likely upgrades to the Sting suite that would improve performance at the cost of greater computational loads. These algorithms are discussed in the next subsection.

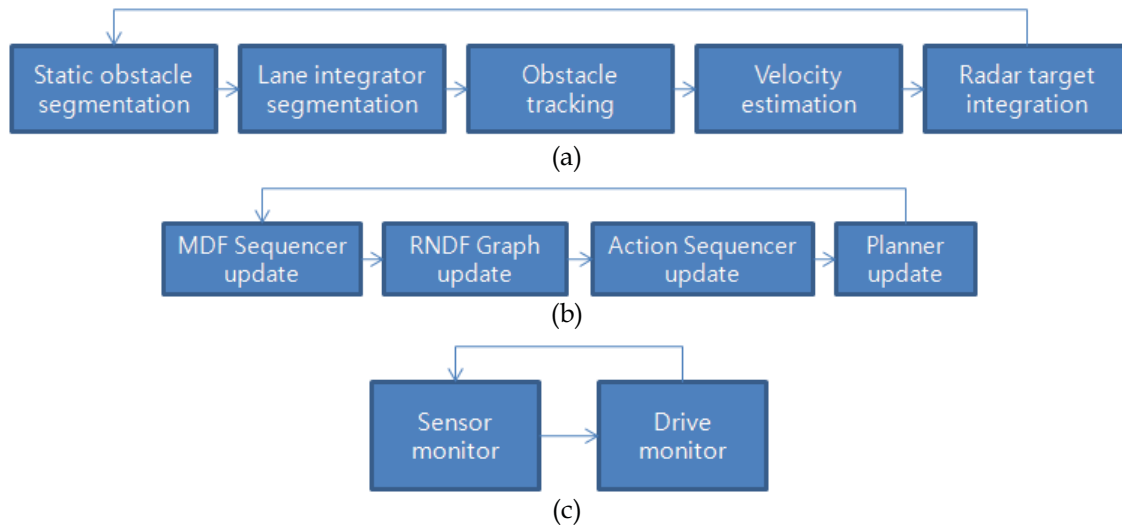


Figure 18. Major steps in the (a) mapper, (b) planner, and (c) controller algorithms.

3.2.4.2 Extreme Scale Autonomous Vehicle Algorithms

GT examined two upgrades to the Sting algorithm suite: a robust land detection and tracking technique [35], and a fast simultaneous location and mapping (“SLAM”) algorithm. These algorithms were chosen to offer a high probability of improved performance, but also to have high computational cost. The intent is to estimate the loads that might be encountered when computational capability was no object, and determine if these rise to the “extreme computing” level.

The major steps in the robust lane detection algorithm are shown in Figure 19. The “lane marking detection” stage includes a support vector machine (SVM) intensity bump classifier. This stage is expected to require about 16 Gops (800×10^6 operations per frame, times 20 fps). The lane boundary hypothesis generation stage is based on a combination of the well-known RANSAC algorithm [36] and a particle filter. Each of these has a complexity that is linear in the number of data points and was therefore assumed not to be a driving factor in the overall computational load.

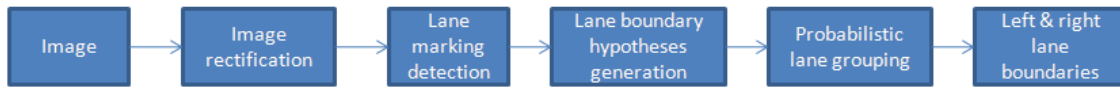


Figure 19. Major steps in the robust lane detection algorithm.

The Fast SLAM algorithm is quite complex; see [37] for details. It is based on a particle filter implementation of an extended Kalman filter approach. The computational complexity is of order $O(N^2P\log N)$, where N is the number of laser range finder detection points, which can be as low as a few hundred to as high as tens of thousands, and P is the number of particles used in the particle filter, which will likely be a few hundred. For $N = 1,000$ and $P = 100$, the estimated operation count is 41.5×10^{10} per frame, giving a total rate of 8.3 Tflops at 20 fps. For a larger problem with $N = 50,000$ and $P = 500$, the estimate becomes 811×10^{12} operations per frame, leading to a computational rate of 16.2 Pflops!

There are many assumptions involved in arriving at these sample computational rate estimates, and it is not clear if all are reasonable. However, they do indicate that it is quite likely that advanced algorithm suites for ground autonomous vehicles could easily achieve extreme scale computational loads. Conversely, if extreme scale computing capability becomes available in embedded form factors, it will enable the deployment of much more advanced machine vision and robotics algorithms that could greatly enhance the capability of autonomous vehicles.

SECTION 4

DATA MOTION ANALYSIS FOR LOW POWER ALGORITHMS

4.1 Locality Analysis of Algorithms

Understanding how an application accesses its data from memory is important in understanding its performance, as accessing successively higher levels of a memory hierarchy comes at an exponentially increasing cost of energy and time. Often, memory accesses of known algorithms occur in a pattern that can be predicted and modeled. By studying these patterns, insight can be gained into how to best minimize energy costs in applications using these algorithms.

On this project, GT began a study of the data movement and memory access properties of key numerical algorithms on modern architectures, with the eventual goal of developing methods to design and optimize algorithms to significantly reduce their energy cost at little to no performance cost, compared to current best-practice algorithms for the same functions. The algorithms considered were two variations each of finite impulse response (FIR) filters, fast Fourier transforms, and matrix multiplication. GT developed a simple method for semi-automatically generating a trace of the data memory access patterns of these algorithms using MATLAB tools, and analyzing that data to obtain information about the application's spatial and temporal locality. These tools were used to calculate quantitative metrics that can be used for making comparison of these different applications.

4.1.1 Locality Metrics

4.1.1.1 Spatial Locality

Spatial locality is an attribute of an algorithm. Given that the algorithm has accessed a particular memory location, spatial locality is a measure of the likelihood that nearby addresses will be accessed by that algorithm in the near future. Applications with high spatial locality are more likely to sequentially access memory locations that are already in a low, local level of the memory hierarchy. Consequently, they will less frequently suffer a cache miss and have to access higher levels of a memory hierarchy and suffer additional delay and energy consumption. High spatial locality is therefore desirable.

To quantify the spatial locality of an algorithm, a memory trace recording the address of each memory access was captured when the algorithm was run. Each memory access in the trace was given a *stride* value, which represents the absolute distance from the current address to the nearest neighbor of the W previously accessed addresses. This "look back window" W is used because considering the stride only between immediately consecutive accesses is likely to miss instances of spatial locality in which local accesses are interleaved. The value of W should be made architecture-specific; for instance, it might be correlated with the size of a local cache line. W should be large enough to capture the locality within application loops, but small enough that the cache availability of the previous accesses is not in question.

A very small example of a relative memory access sequence and the corresponding stride calculation, using a very small value of $W = 4$, is shown in Table 9. Consider time step 3. Even

though the stride from the access at time step 2 is 7 locations, the stride from the access at step 1 is only 1 location. Since step 1 falls within the look back window from step 3, the stride is considered to be 1 at time step 3. Note that the first access in a trace will not have a stride.

Table 9. Example of spatial locality stride calculation with $W=4$.

Time Step	1	2	3	4	5	6	7	8	9	10
Address	0	8	1	9	7	15	4	10	10	7
Stride	n/a	8	1	1	1	6	3	1	0	3

Given this array of stride values for each access, a single-number spatial locality score can be defined [38]. This score is based on the formula below, where is the fraction of total non-zero stride memory operations that are of stride length i .

$$L_s = \sum_i \left(\frac{\text{stride}_i}{i} \right) \quad (5)$$

This definition results in a score that is normalized to a range of [0,1] and is inversely proportional to the average length of strides. Thus, a trace with all minimum strides, length $I = 1$, will get a perfect score of $L_s = 1$, whereas a trace consisting entirely of large strides will have a score that approaches zero. Scores closer to 1 are expected to represent algorithms with lower data motion and energy costs. As an example, the following calculation gives the spatial locality score for the data in Table 9:

$$L_s = \frac{4/8}{1} + \frac{2/8}{3} + \frac{1/8}{6} + \frac{1/8}{8} = 0.6198 \quad (6)$$

4.1.1.2 Temporal Locality

Temporal locality is the degree to which an algorithm exhibits the phenomenon that, when a memory location is accessed, it is likely that the same location will be accessed again in the near future. Similar to spatial locality, applications with high temporal locality are less likely to suffer cache or other local memory misses and thus have to access higher levels of a memory hierarchy. High temporal locality is therefore also desirable.

To quantify the temporal locality of a memory trace, each memory access was assigned a *reuse distance* that represents the number of unique memory accesses since that location's most recent access. Table 10 is an example of computing reuse distance for a sample memory trace [39]. For example, memory location d is accessed at time steps 1 and 7, and has three unique accesses between the two: a , b , and c . Note that the first access of a memory location in a trace will not have a reuse distance.

Table 10. Example of reuse distance calculation.

Time Step	1	2	3	4	5	6	7	8	9	10	11	12
Address	<i>d</i>	<i>a</i>	<i>c</i>	<i>b</i>	<i>c</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>a</i>	<i>f</i>	<i>b</i>
Reuse Distance	n/a	n/a	n/a	n/a	1	0	3	n/a	n/a	5	1	5

While this simple example has only a few memory addresses to track, the calculation of reuse distance for larger traces can become computationally intensive due to the need to store a “last used” value for every memory location, as well as iterating through each of these “last used” values for each memory access. One possible solution to speeding this calculation up is to keep the “last used” values in a binary search tree to reduce the time to calculate each reuse distance from $O(M)$ to $O(\log M)$ for a balanced tree [39].

Another solution is to approximate reuse distance into “bins” instead of exact values. This solution works well with the single value temporal locality score presented below, which uses binned values to calculate the score. The bin sizes should be associated with the characteristics of a typical memory hierarchy, such as those shown in Figure 20.

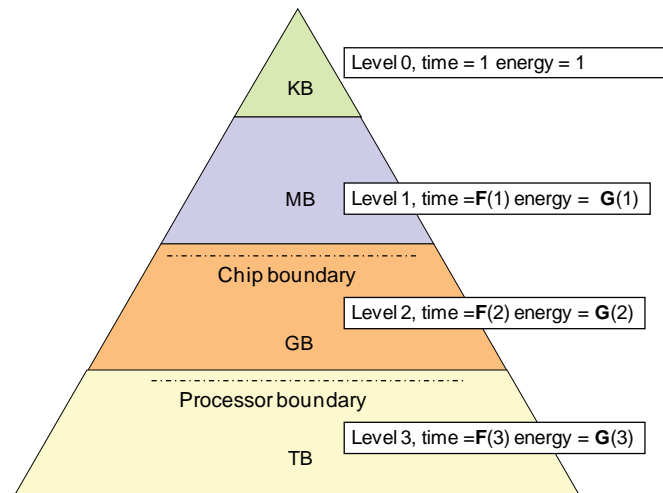


Figure 20. Abstract memory hierarchy for binning locality scores.

In this hierarchy, every level has a capacity in bytes. The capacity grows as $base^{level}$; in the figure, the base is 1000. The levels and their capacities cross some architectural boundaries dictated by available processing technologies (on chip, on processor, on machine etc.). The functions describing the time and energy to access an element at each level are of interest for the data motion metric discussed in Section 4.2, but are not needed for this discussion of temporal locality. Similar to spatial locality, the array of reuse distances allows computation of a single-

number temporal locality score [38] as defined in Eq. (7). In this equation, $\frac{reuse_{2^{i+1}} - reuse_{2^i}}{\log_2 N}$ is the fraction of dynamic memory operations with reuse distance less than or equal to i . N could be the maximum reuse distance of a trace or some arbitrary value of reuse distance used to establish a standard between traces.

$$L_T = \sum_{i=0}^{\log_2 N - 1} \frac{(reuse_{2^{i+1}} - reuse_{2^i}) \cdot (\log_2 N - i)}{\log_2 N} \quad (7)$$

As defined, this score is also normalized to a range of [0,1] with one being perfect locality and zero being extremely poor temporal locality (large reuse distances). It is desirable to have temporal locality scores as close to one as possible. However, a problem with this definition of L_T is that its “binned” nature produces results that aren’t necessarily intuitive. This can be seen in Figure 21, which shows the variation of L_T for basic matrix multiplication vs. access sequence length N . Even though the size of the matrix multiplication is increasing, the temporal locality is not decreasing in a smooth fashion. This result is not in error, but further analysis is required to determine whether this definition of L_T is adequate for the overall goal or if refinements are needed.

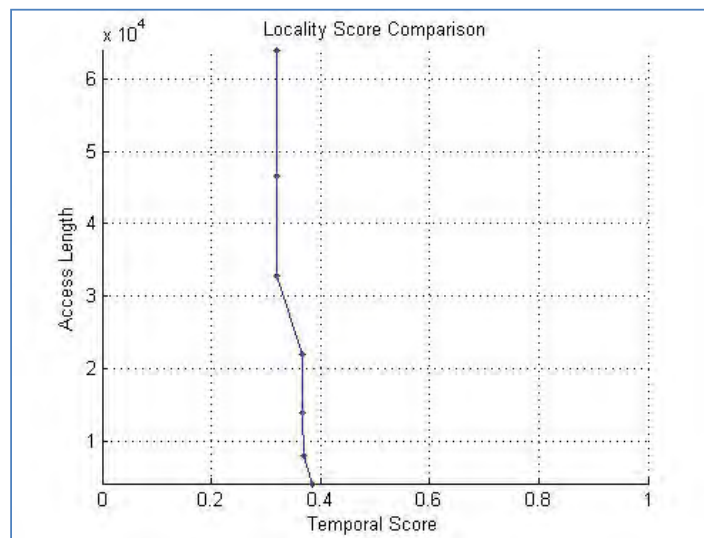


Figure 21. “Binned” temporal locality L_T vs. trace length for a basic matrix multiplication algorithm.

4.1.2 Locality Analysis Tools

Initially, memory traces were generated using MATLAB programs written to explicitly compute memory access patterns for the selected algorithms. The MATLAB code did not actually compute the function of interest. Instead, it generated the series of relative memory addresses, or “trace”, that would be generated by a sequential program running that function. This sequence was predicted based upon either or both of a mathematical formula for the function

computation (e.g., a convolution equation for the FIR filter) or a flowgraph (for the FFT). This memory trace could then be used to compute and analyze the algorithm's locality.

While the initial method of generating traces provided a useful baseline for calculating locality scores, GT subsequently implemented a more robust, accurate, and semi-automated method of generating these trace patterns. A MATLAB class was created that can be used in algorithm computation and is able to capture the memory address sequences, subject to some limitations on how the algorithm is expressed in MATLAB. Any array or matrix can be defined as a `traceable` class in MATLAB (or `ctraceable` for complex data). Then, any read or write access to the data within that array causes the details of that access to be printed to the MATLAB command window, where it can be captured and subsequently analyzed for locality computations. With this capability, it is not necessary to write new functions to generate traces explicitly. Instead, any existing function that computes the mathematical function of interest can be used with only minor modification.

This process can be thought of much like loading and storing to memory with assembly level code. The data is loaded from a `traceable` class (interpreted as a `read`), manipulated locally, and then stored back to a `traceable` class (interpreted as a `write`). A simple example of this is shown below in the form of a scalar multiplication operation $X=X*Y$:

```
X = traceable(1);
Y = traceable(1);
regA = X(1);
regB = Y(1);
regA = regA * regB;
X(1) = regA;
```

This sequence produces the following text output to the MATLAB command window, where the first column is a read/write flag denoted R/W (0 for read, 1 for write); the second column is the relative memory address of the array being accessed; and the third column is the index of the access within that array.

```
0 8904 1
0 8905 1
1 8904 1
```

The exact value of the array memory address is somewhat arbitrary; it represents an internal counter that keeps different arrays from overlapping during runtime. Using the MATLAB `clear` command prior to running the algorithm will reset the array addresses to begin at zero. However, this is unnecessary, as only the relative addresses are used during stride computation.

A limitation of the `traceable` class is that, currently, it does not interact with many MATLAB functions that do not make explicit assignments or references. For example, `traceable` arrays cannot be input or output variables from a function, which means that recursive functions must be unrolled. Another limitation of `traceable` is that the trace pattern operates under the assumption that all arrays are adjacent in memory, which usually is not the case. However, this

difference does not produce significantly different spatial locality scores due to the stride detection window, as well as the manner in which very large stride lengths are handled in the computations. Future research will improve the traceable class as required to expand its usefulness.

4.1.3 Spatial and Temporal Locality Experiments

Multiple example algorithms with various configurations were tested. These were initially tested using the “manual” method of generating trace patterns, but most were adapted for use with the `traceable` functionality to ensure that the resulting data was as close as possible to real world situations. The traces resulting from running these algorithms through the `traceable` class were examined closely to ensure that they produced the expected patterns for basic algorithms with known access sequences.

Two implementations of the radix-2 FFT were compared: one where the elements of the input array were bit-reordered, which allowed for a simplified method for computing each state, and one where the elements were presented in natural order, which used a more traditional FFT algorithm [40]. Input sizes were limited to powers of two for simplicity. As shown in Figure 22, the bit-reordered version of the FFT demonstrated noticeably higher spatial locality scores, while the temporal locality scores remained consistent between the two. Generally, as the input size increased for an FFT, both the spatial and temporal scores decreased, with spatial locality being affected by the input size to a greater extent.

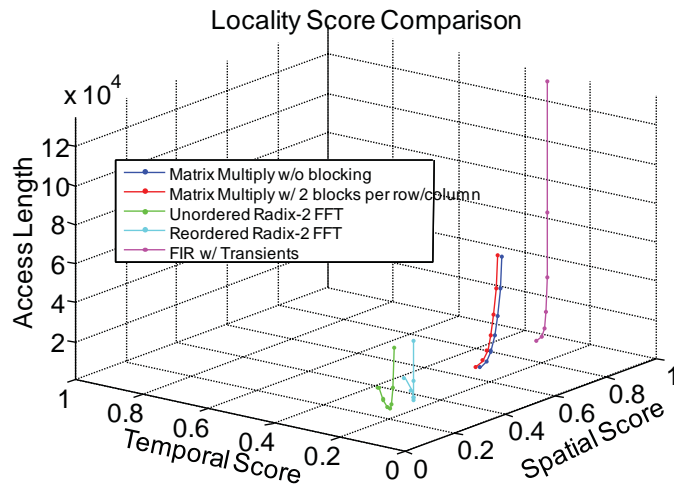


Figure 22. Temporal and spatial locality vs. algorithm size.

A matrix multiplication function was implemented, capable of both standard and block matrix multiplication. For simplicity, input matrices were square and consisted of an integer number of square blocks [41]. As shown in Figure 22, the spatial locality score remained fairly consistent

across various input sizes and block sizes. In addition, the blocked matrix multiply demonstrated consistently higher temporal locality scores over non-block multiplications. This result is as expected, as the data in each block of the matrix is used multiple times in succession before the algorithm moves on to the next block.

Finally, an FIR filter algorithm was implemented both with and without transient endpoint calculations. Of these three different algorithms, in general the FIR filter has the best spatial locality score and the worst temporal locality score; the FFT had the best temporal locality score and the worst spatial locality score; and the various matrix multiplications occupied a middle ground for both locality scores. These results are in line with would be intuitively expected from the simulation, such as blocking a matrix increasing temporal locality and ordering an FFT inputs increasing spatial locality.

4.2 Data Motion Metric

A more comprehensive metric proposed to quantify the energy cost of a particular application is a *data motion metric* (DMM). This metric was an attempt to obtain a simple approximation of the energy cost involved in moving the data for each memory access, summed across the entire memory trace to obtain the total energy cost. It builds on the spatial and temporal locality metrics to estimate data motion cost according to

$$DMM = \sum \min(L_T, L_s) \quad (8)$$

This formula could give a basic approximation for the relative total energy cost of all of the memory accesses in a given execution. A drawback of this metric is that it gives equal weight to temporal and spatial locality, whereas it is believed in modern architectures that the greatest data motion cost occurs in vertical movements through the memory hierarchy. *DMM* was not evaluated during the period of this project.

Although not evaluated during this project, an extension of the DMM idea has been proposed by A. Snively. This extension bins all accesses into the memory hierarchy with associated cost functions F and G as discussed above. The level of the accesses, averaged across all accesses, is computed. This average access level is then divided by the spatial locality L_s . The time and energy to access an element of Level 0 is normalized to 1. The time to access a level other than 1 is a (possibly piecewise) function F of the level, while the energy to access a level is a (possibly piecewise) function G of the level. Snively has proposed $F = 10^{\text{level}}$, which is somewhat arbitrary but does yield some plausible latencies in the context of today's technologies. W. Dally has proposed a function of the following general form for G :

```

If capacity(level) < chip boundary
    G = 1 + SQRT(capacity)
Else If capacity(level) < processor boundary
    G = 1 + LARGE + SQRT(capacity(level))
Else
    G = LOGbigbase(capacity(level))

```

The resultant metric may be closer to the desired single metric for data motion cost comparisons.

Figure 23 shows values of the DMM computed for variations of Fourier transforms, FIR filters, and matrix multiplication. The FIR filter scores nearly identically with and without end transient calculations, as would be expected. Somewhat more surprisingly, so do the blocked and unblocked matrix multiplies. Three variations of the Fourier transform are shown: the “brute force” discrete Fourier transform (DFT), a standard radix-2 Cooley-Tukey FFT, and a “re-ordered” FFT that eliminates the bit-reversed reordering step of standard FFTs, which requires many non-unit-stride accesses [40]. The DFT has a low data motion cost, but also provides poor performance. The FFT, which radically improves performance, unfortunately demonstrates a high value of *DMM*. However, the re-ordered FFT, which has essentially the same performance as the standard FFT, also achieves the low data motion cost of the DFT. This example demonstrates the potential for finding algorithms that combine high performance and low data motion cost that motivates this research.

Although this initial demonstration was intriguing, there was not sufficient time on this project to systematically investigate the DMM. GT will investigate this and other means of characterizing and designing low-energy, high performance algorithms in future research.

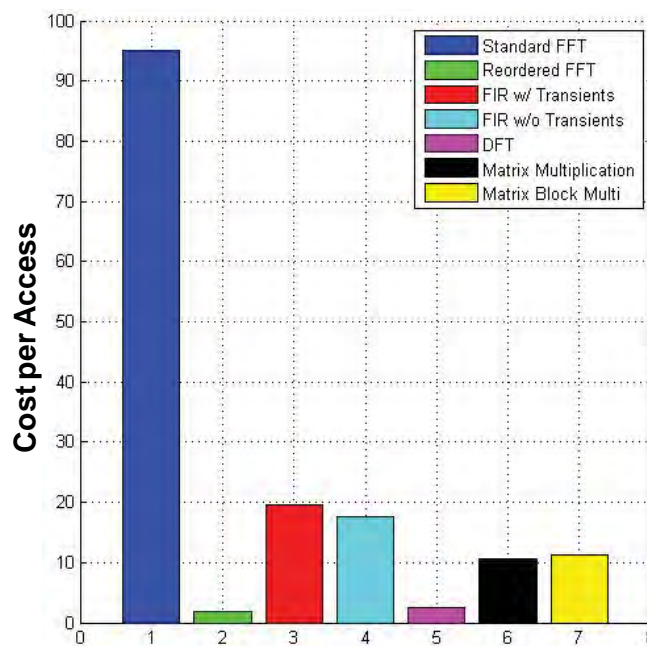


Figure 23. Data motion metric for several numerical algorithms.

4.3 Data Motion Analysis Conclusions

Energy and power costs are a major constraint on the development of modern EES systems, and a large part of energy use is in accessing data from memory. It is hoped that by gaining a better understanding of temporal and spatial locality and their impact on data motion within a memory hierarchy, it may be possible to design advanced algorithms to reduce these energy

costs. By tracing memory access patterns, the tool infrastructure described here is able to quantify the impact of different algorithmic realizations of a numerical function on locality and thus, it is hoped, on data motion and energy efficiency. For example, it can be seen that matrix blocking is an algorithmic technique that can be used to improve temporal locality and, as a result, possibly reduce algorithm energy costs.

Much work remains to build from these initial demonstrations to a practically useful methodology. It is critical to establish and validate the relationship between locality and data motion scores and actual energy costs of an algorithm on a modern architecture. For example, how does increasing the temporal locality by 20% affect the energy efficiency of a particular application? Another area of exploration is the effect of multithreading and the use of multiple cores on a given algorithm's locality, and whether specific architectures can be mapped to the calculations to result in more accurate and less generic scores. Yet another is scale. The examples given here are very small "toy" problems used to establish basic concepts and definitions. Do the proposed methods and metrics scale to large programs?

Finally, the analysis of locality and data motion, and the prediction of algorithm energy efficiency based on these analyses, is not an end in itself. Rather, the intent is to provide the basis for developing design techniques that lead to high performance numerical algorithms with inherently low data motion costs. Another avenue of research is in ways to design such algorithms. One possibility would be to develop methods for analysis of mathematical expressions, perhaps in a factored matrix form, of different algorithms for the same function. Another would be to apply the technology of autotuning, used primarily for improving algorithm speed so far, to the minimization of data motion or energy subject to a minimum performance constraint.

SECTION 5

HIGH PERFORMANCE LIBRARIES FOR MULTICORE PROCESSORS

5.1 Background

During the course of this project, the use of graphical processing units (GPUs) for “general purpose” scientific computing, including signal processing, has continued to grow rapidly in interest and acceptance in the high performance computing community. GPUs are proving popular platforms for experimentation for high performance computers ranging from embedded systems of a few cards, to some of the fastest computers in the world [24]. Figure 24 illustrates the growth in GPU single precision floating point performance from 2001 to 2010, a trend that has continued into 2011. Note that the fastest GPUs now offer in excess of one teraflops performance in a single chip!

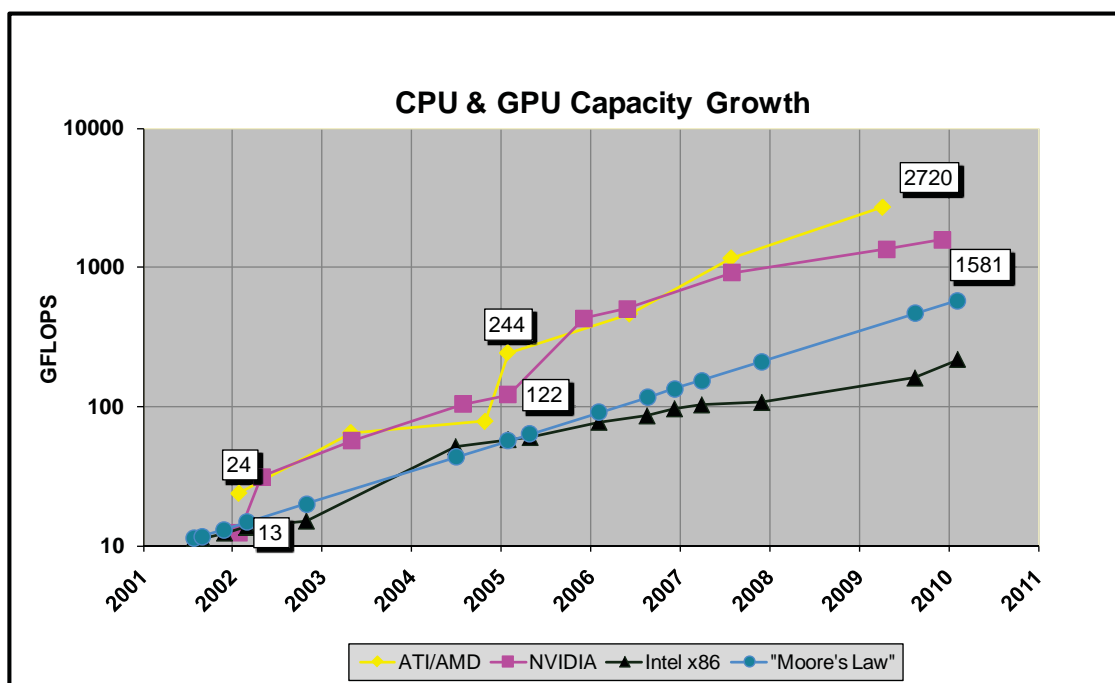


Figure 24. Growth of GPU single precision floating point performance.

GPUs are many-core devices. This fact, coupled with the rapid performance increases, relatively low cost, and improving toolsets has led Georgia Tech to investigate their applicability to defense-relevant high performance embedded computing under a series of high performance computing projects, including this one. Specifically, Georgia Tech's work with GPUs under this project has focused on continued development of GPU middleware based on the VSIPL (Vector, Signal, and Image Processing Library) application programming interface (API).

5.2 VSIPL

VSIPL is a portable API for implementing high-performance signal processing applications while retaining platform independence. The API specification document, currently at version 1.3, is available at the VSIPL web site [42]. VSIPL supports memory abstractions for utilizing coprocessors with disjoint memory spaces. A signal processing application may structure input data in a *block*, *admit* it once to VSIPL's memory management, perform computations on that data, and *release* only the block containing the final result. Intermediate results are not transferred between system and GPU memory, avoiding unnecessary latencies and communications overhead. This capability distinguishes VSIPL from other signal processing libraries that permit random access to data.

The initial cost to an implementer of developing a complete VSIPL library can be substantial. Consequently, VSIPL defines two *profiles*, VSIPL Core and VSIPL Core Lite. Each defines a reduced, but very useful subset of the full VSIPL functionality that can be implemented at lower cost. Documents defining the exact contents of each profile are available at [42]. Generally, the Core Lite profile supports single-precision real and complex-valued vectors, and provides the following functionality:

- ◆ Support functions for creating, modifying, and destroying blocks and managing associated buffers;
- ◆ Element-wise mathematical operations;
- ◆ Inner products and scalar operations;
- ◆ Extrema searching;
- ◆ FIR filtering;
- ◆ Out-of-place Fast Fourier Transform (FFT); and
- ◆ Histogram and portable random number generator.

The Core profile adds support for single-precision real and complex-valued matrices, and provides the following additional functionality:

- ◆ Matrix element-wise operations;
- ◆ Window creation;
- ◆ Convolution and correlation;
- ◆ Elementary linear algebraic operations (transpose, matrix-vector products and sum variations, etc.); and
- ◆ Linear equation solvers.

In recent years, active development of the VSIPL specification has focused on defining a C++ binding known as VSIPL++. This binding provides parallel and object-oriented extensions to VSIPL with the goal of unifying computation and communication in a single high performance, productive, and portable API. Most of the work on extending VSIPL to VSIPL++ has been done by the VSIPL Forum operating in conjunction with the High Performance Embedded Computing Software Initiative (HPEC-SI) [43]. An API specification has been developed for VSIPL++ version 1.02, with a version 1.03 draft pending [43].

5.3 GPU VSIPL

With NVIDIA's release of the CUDA environment in 2007, Georgia Tech began development of a more complete and high performance implementation of C VSIPL for NVIDIA GPUs. That effort has continued under this contract, and has culminated in development of the GPU VSIPL Library [44].

GPU VSIPL is an implementation of VSIPL that, at this writing, now includes most of the VSIPL Core functionality, with the exception of some of the linear equation solvers and random number functions, but with the addition of a large number of matrix arithmetic operations not included in VSIPL Core profile. It passes all compliance tests of the VSIPL Test Suite [42]. GPU VSIPL is implemented with NVIDIA's CUDA programming language and C++ compiled with Visual Studio 2005, and has been run on versions of CUDA from 2.3 through 3.2 (the current version at this writing) and on a variety of NVIDIA GPUs, including those current as of late 2010. Details of the implementation techniques for the library are given in [45].

GPU VSIPL is freely distributed as a static binary library with C linkage at the GPU VSIPL web site [44]. Georgia Tech also licenses the GPU VSIPL source code.

The GPU VSIPL web site provides a sample radar range-Doppler map calculation application, intended as both a demonstration and a tutorial application for new users of VSIPL and GPU VSIPL. The basics of range-Doppler processing are described in [46]. In brief, range-Doppler mapping is a radar signal processing technique, common in both ground and especially airborne radars, which separates the echo energy of multiple radar targets from one another based on the Doppler shift of the targets, which is related to their velocity, and their time delay, which is related to the distance (range) to the target. It also amplifies the target echoes relative to the system noise, making target detection easier. Figure 25 shows the output of this application.

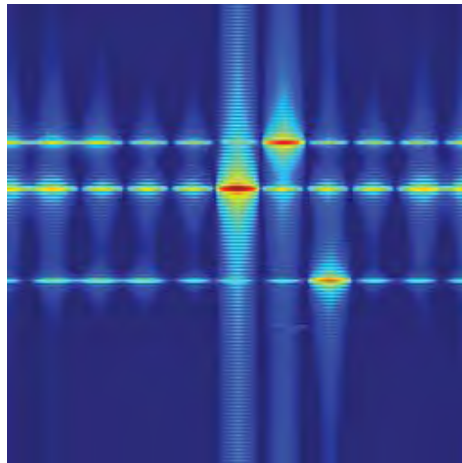


Figure 25. Output of GPU VSIPL range-Doppler map application showing three targets with sidelobes in both range (vertical) and Doppler (horizontal) dimensions.

Computationally, a 2D matrix of complex-valued floating point data is the input to the process. Independent 1D FIR filtering operations with complex coefficients are performed on each column of the data. Independent 1D FFTs are then performed on each of the rows. Finally, the magnitude of the result is calculated, usually on a decibel scale, and displayed. Variations involve the use of time- or frequency-domain convolution for the FIR operations, and the use (or not) of windows for sidelobe control in both dimensions. Using a GeForce GTX 280 and GPU VSIPL, a speedup of 75x was obtained compared to an implementation using the TASP VSIPL Core Plus library on a 2.83 GHz Intel Core 2 processor.

5.4 Progress in the Development of GPU VSIPL

Under the EES study portion of this project, development of GPU VSIPL has concentrated on extension of the library to include important matrix decompositions and linear equation solvers from the Core profile, and update of the VSIPL Test Suite.

The matrix decomposition and equation solver work has focused on addition and debugging of the singular value decomposition (SVD) and QR decomposition (QRD) families of functions to the library, and the Toeplitz system linear solver. For the SVD, the functions `svdprodu`, `svdprodv`, `svdmatu`, and `svdmatv` were added to the library. The handling of various input cases by the main decomposition function of the SVD algorithm has been debugged and validated. However, the final decomposition results, while close to expected values, do not currently meet the validity specifications. Regarding the QRD functions, bugs in the existing real-valued QRD function were fixed. Development of the complex QRD functions, as well as the real and complex Toeplitz solvers, is ongoing.

The functionality of the VSIPL Test Suite [47] was validated against the current version TASP (Tactical Advanced Signal Processor) VSIPL Reference Implementation [42]. This comparison led to corrections or additions of data to the test suite for the VSIPL functions `toeplitz`, `ctoeplitz`, `qrd`, `cqrd`, `lud`, `cmprod`, and `cmprodt`. New test suite functions were written for validation of the SVD-related VSIPL functions. Finally, an ongoing effort was begun to add the element-wise functions present in the VSIPL Core profile, but not in the Core Lite profile, to the test suite.

Work to debug and validate the SVD functions; complete and validate the QRD decomposition and Toeplitz solver functions; add other solvers from the Core profile; and update the Test Suite as required will continue under other projects.

SECTION 6

METRICS FOR EXTREME SCALE SYSTEMS

6.1 Introduction

UHPC systems, which include EES systems at the embedded scale, will be characterized and evaluated by a number of metrics. The UHPC program is considering the development of metrics to characterize the following aspects of UHPC systems:

- ◆ *Performance* – UHPC systems must provide breakthrough performance across application workloads including the UHPC challenge problems.
- ◆ *Scalability* – UHPC systems are expected to provide an unmatched level of parallelism, far beyond the current levels of scalability.
- ◆ *Energy efficiency* – UHPC systems face extreme challenges in energy efficiency.
- ◆ *Dependability* – UHPC systems must adapt and continue to operate under continuous occurrence of faults.
- ◆ *Resiliency* – UHPC systems must be resilient, having the ability to continue through faults and cyberthreats.
- ◆ *Security* – UHPC systems must be secure, with cyber protection intrinsic to the entire architecture from hardware on up.
- ◆ *Self-awareness* – UHPC systems must be adaptable and flexible, responding to dynamic changes in workload and environment.
- ◆ *Productivity* – UHPC systems must be productive for their customers.

Each of these aspects will be evaluated using one or more metrics. Metrics for some of these aspects are well known. Performance can be evaluated using time-to-solution for workloads at various scales, from micro-benchmarks to challenge problems and full-scale applications. Energy efficiency can be characterized with measurements of actual energy consumption at multiple levels, potentially from component level through full system level. Scalability (weak and strong) can be measured by increasing the size of a mixed workload, increasing the number of instance of a problem on separate data sets, increasing the size of a single data set, and increasing the number of processors on a fixed size data set.

Other aspects are harder to characterize. There are no agreed-upon metrics for resiliency or self-awareness, for instance. While the High Productivity Computing Systems (HPCS) program addressed productivity, it has not to date resulted in widely-accepted productivity metrics. In the UHPC program, it is expected that in the near term the emphasis will be on programmability rather than a broader definition of productivity. A system is considered highly programmable if it does not require application programmers to explicitly manage system complexity in order to achieve their performance and time to solution goals. Programmability is important in determining the utility and productivity of a high performance system. While architectural features and optimizations may be available, the ease of leveraging the optimizations will largely dictate whether a programmer is able to use the features effectively or at all.

In this project, GT has investigated techniques for assessing the programmability of high performance parallel systems. Based on this investigation, an initial proposal for a methodology to evaluate programmability has been developed and is described in the sections that follow.

The proposed methodology would evaluate a system based upon the difficulty of developing programs for that system to implement specific standard parallel algorithms representing certain “programming patterns”. These patterns serve as proxies for groups of applications having similar programming requirements. The programs developed will be scored with regard to a set of “cognitive dimensions” that attempt to recognize the human difficulty in developing the source code. Finally, the scores for each standard program will be combined into an overall programmability score.

6.2 Measuring Programmability

6.2.1 Cognitive Dimensions and Programmability

Research has been done in the past to discern what makes programming “hard” or “easy” for certain problems and environments. In [48] a set of categories for programmability were proposed called the “cognitive dimensions” for programming. In [49], Mattson adds two more dimensions relevant to parallel programming. This augmented set of cognitive dimensions is outlined in Table 11 with brief descriptions of the meaning of each one.

Table 11. Cognitive Dimensions

Cognitive Dimension	Description
Viscosity	Difficulty in introducing small changes
Hidden Dependencies	Does a change in one part of the program cause other parts to change that is not overtly apparent in the text?
Error Proneness	How easy is it to make mistakes?
Progressive Evaluation	Can you check a program while incomplete? Can parallelism be added incrementally?
Abstraction Gradient	How much abstraction is required/possible?
Closeness of mapping	How well does the problem map to the architecture?
Premature Commitment	Does notation constrain the order? E.g. forced look ahead
Consistency	Does similar syntax imply similar behavior?
Hard mental operations	Complexity of primitive operations
Terseness	Succinctness
HW visibility	What features are available to the programmer?
Portability	Is code specific to a certain model of the architecture?

The programmability methodology will examine each of these dimensions for each of several programs implementing different parallel patterns to develop the overall programmability score.

The cognitive dimensions framework arises from the human-computer interface (HCI) community and is useful in analyzing notations, user interfaces, and programming language design. A reservation to this approach is that, at this writing, it is unclear how strong a connection has been established between high cognitive dimension scores and ease of programming of complex problems.

6.3 Parallel Programming Patterns

Parallel programming encompasses a broad class of problems and algorithms, each with their own challenges to correct and efficient programming. However, nearly all parallel programs follow one of the many standard “patterns” for developing parallel programs [50],[51]. Parallel patterns occur at all levels of the algorithmic and software stack, ranging from high level concurrency models such as pipe-and-filter, to algorithmic patterns such as task parallelism and geometric decomposition, to low level communication patterns such as single program multiple data (SPMD), master/worker, or shared memory. In Figure 26 a compilation of various patterns is shown from the OPL 2.0 [52]. OPL (Our Pattern Language) is an evolving collection of the most common patterns for parallel programming and is the result of ongoing parallel programming research at the University of California, Berkeley (UCB).

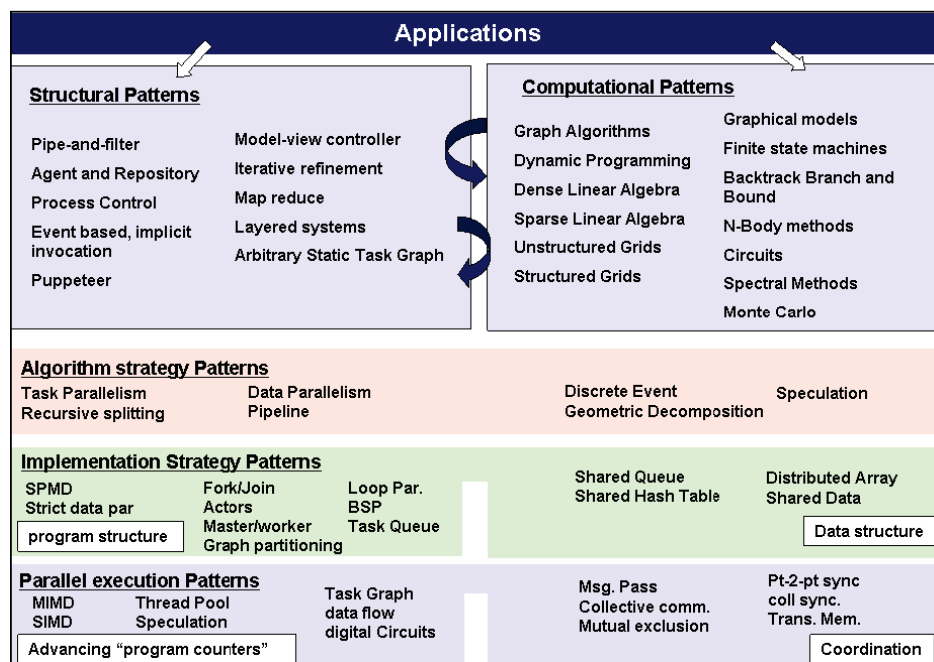


Figure 26. OPL 2.0 Patterns.

Patterns at each level represent a unique programming style. For example, at the algorithm level, task parallel is a pattern used when the concurrency exposed is from several independent

tasks working on independent data. Therefore, task parallel is classified as being easily parallelizable and benefiting from low overhead for spawning and joining threads. This differs from another common pattern, geometric decomposition, which concerns parallelism resulting from splitting a large data set into subcomponents and working on each subcomponent in parallel. This pattern is classified by the sharing that occurs typically between adjacent data chunks, particularly the “edges” of the data.

A good programmability metric should be applicable to a broad class of potential applications of the system being evaluated. While some full-scale applications may be dominated by one parallel pattern, others may incorporate involve several. As an example, molecular dynamics codes such as the UHPC program’s challenge problem #5 include elements of the n -body, geometric decomposition, master/worker, and shared memory patterns. Rather than try to directly measure the programmability on a number of major applications, each of which in turn may be composed of a number of sub-parts having significantly different characteristics, the proposed methodology focuses on measuring programmability of a set of programs that represent common parallel pattern groupings. In effect, the test programs and the parallel patterns they incorporate are used as a set of “basis programs” for “spanning the space” of larger or more diverse applications. Measuring the programmability of the basis programs on a given system can, hopefully, be used to infer the programmability of the system on other applications. For instance, the n -body score would apply to the Barnes-Hut or fast-multiple method algorithm; the geometric decomposition score would apply to a broad class of algorithms such as ocean simulations, blood simulations, or heat dispersal.

6.3.1 Standard Parallel Problems

Ongoing research in parallel computing at UCB has developed a set of 13 canonical parallel programs, listed in Table 12 [53].³ Each implements a set of communication and computation patterns that are common in real applications. Figure 27 illustrates how a number of specific applications and general application areas rely in different degrees on various of these patterns [53]. Furthermore, each program typically incorporates several of the parallel patterns described above. Collectively, these 13 programs are representative of the combinations of parallel programming patterns needed to program real applications on UHPC and EES systems.

Table 12. Thirteen Canonical Algorithms.

Dense Linear Algebra	Sparse Linear Algebra
Spectral methods	n -body methods
Structured grids	Unstructured grids
Map Reduce	Combinatorial logic
Graph traversal	Dynamic programming
Back-track/branch and bound	Graphical methods
Finite State Machines	

³ These programs are sometimes referred to as the 13 “dwarfs”.

	Embed	SPEC	DB	Games	ML	CAD	HPC	Health	Image	Speech	Music	Browser
1. Finite State Mach.												
2. Circuits												
3. Graph Algorithms												
4. Structured Grid												
5. Dense Matrix												
6. Sparse Matrix												
7. Spectral (FFT)												
8. Dynamic Prog												
9. Particle Methods												
10. Backtrack/B&B												
11. Graphical Models												
12. Unstructured Grid												

Figure 27. "Temperature chart" indicating reliance of various application codes on the 13 canonical parallel programs. From [53].

6.4 Proposed Methodology

Prior efforts have measured programmability by providing a group of programmers with problems to code and then evaluating their efficiency through such measurements as development time and keystrokes, and with post-session surveys. Here a different methodology is proposed based on scoring a set of parallel codes written by the system developers are scored on each of the cognitive dimensions, and an overall programmability score is derived from those results.

More specifically, the proposed methodology proceeds as follows. "Developer" refers to the developer of a UHPC system to be evaluated, and of the source code programs that are the basis for that evaluation. "Evaluator" refers to a second party that performs the evaluation and assigns the programmability score for the developer's system.

1. The evaluator will be provided with a standard set of the 13 parallel pattern problems and a parallel reference implementation code for each.
2. The developer will write a separate parallel program for each problem. The programs need not be parallelized in the same way as the reference implementation, but should instead be parallelized in ways that make best use of the developer's system. The source code for these programs will be provided to the evaluator, along with any requested

statistics regarding development of the source and executable code, and the performance of the executable code on the developer's system.

3. The developer will also provide the evaluator with an API document describing the programming environment used and a computer system architecture specification.
4. The evaluator will analyze the 13 codes using a standard measurement system to score the effectiveness of each program in each cognitive dimension. The measurements will be automated and quantitative wherever possible.
5. For each code, the individual cognitive dimension scores will be normalized and a weighted combination formed to create a programmability measure which scores the machine's effectiveness at programming each parallel pattern.

While the programmability scores for each pattern could be further combined to provide a single score for the system, this is not proposed. Some UHPC or EES systems may be designed to be particularly effective on certain classes of problems, at the cost of reduced performance on certain others of less interest to that developer. A single score would obscure these differences and penalize the developer of a specialized machine unnecessarily.

In addition to the standard problems, the evaluator will also be responsible for providing the following items to the developer:

- ◆ Any rules and constraints on the coding exercises;
- ◆ Any parameter sets and data sets needed to run the problems;
- ◆ A means to validate correctness of each program; and
- ◆ Definition of all requested documentation, and development and performance statistics.

A major step in implementing this methodology is defining objective ways to score a program on each of the cognitive dimensions. Desirable characteristics are that the score for each dimension be quantitative and automated, so that scoring is objective and consistent. No standard way to score the cognitive dimensions is known. Table 13 proposes an initial set of measurements for 11 of the 12 cognitive dimensions. (Portability is not scored on the assumption that these codes are developed for a specific system, so that portability is not a primary concern.) This table suggests automatable measurements for eight of the 11 dimensions shown. Further research is needed to determine if two more can be automated. Progressive evaluation, typically measured by determining if a program still runs when subfunctions are replaced with stubs, does not appear to be amenable to automation. This could be evaluated by the developers using a protocol specified by the evaluator, or could simply be excluded from the programmability metric.

Table 13. Potential for Automated Measurement.

Cognitive Dimension	Potentially Automated? / How?
Viscosity	Yes – Length + use of non-standard symbols + depth of nesting + number of special functions needed
Hidden Dependencies	Yes – Measure global accesses
Error Proneness	Yes – Potentially measure the error proneness of certain directives then measure the directive frequency of use
Progressive Evaluation	No
Abstraction Gradient	Unknown – need to measure abstraction exposedness
Closeness of mapping	Yes – Parallel pattern methodology addresses this
Premature Commitment	Yes – Require multiple versions of the code at various stages
Consistency	Yes – Variation between patterns and similarity among patterns
Hard Mental Operations	Yes – Pointer manipulation count
Terseness	Yes - length
HW visibility	Unknown – Related to abstraction

6.5 Experiments

In developing the proposed programmability metric, GT conducted two small experiments to help define and demonstrate this initial methodology. The procedure for these experiments was as follows:

- A well-known algorithm was selected, and a serial algorithm for each was obtained or developed.
- Two parallel programs were written for that algorithm, one using OpenMP and one using Pthreads. The programs were compiled and executed on a very small machine, an Apple Macintosh laptop.
- For each parallel program, a subset of the cognitive dimensions was scored using the measurements suggested in Table 13.
- The scores for each dimension were normalized to a range of zero to one, with one representing a better score.
- Finally, the cognitive dimension scores relating to a program were averaged (using uniform weights) to create the programmability scores for that program.

6.5.1 Example 1 - Quicksort

The divide-and-conquer canonical program was evaluated using a parallel quicksort algorithm. The results of the analysis are shown below in Table 14. Figure 28 displays six of the normalized scores for each API in a Kiviat diagram. This diagram shows that the OpenMP implementation scores very high on the “hard mental” dimensions. This occurred because the conversion of the small serial code to OpenMP was essentially trivial in this very small example. The Pthreads implementation required significantly more work to convert. On the other hand, the same ease of parallelization in OpenMP abstracts away from the user the resulting parallel dependencies, while the Pthreads implementation makes them much more explicit, resulting in a much higher score for “hidden dependencies” for Pthreads than for OpenMP.

Table 14. Measurements for Quicksort program.

	Pthreads	OpenMP
Char Count	2897	2159
Line Count	143	127
Parentheses	70	42
Braces	10	17
API Calls	5	7
Progressive Evaluation	1	1
Nesting Level	1	3
Argument Count	12	2
API Ordering Constraints	2	4
Pointer Manipulation Count	22	1

Overall, these results indicate that the Pthreads version excelled in abstraction, hidden dependencies, and viscosity, whereas the OpenMP version excelled in terseness and hard mental operations. Figure 29 shows the programmability score obtained by uniformly averaging all of the individual scores for each API. The average overall score gives a slight edge to the OpenMP version. The programmer who wrote both codes concurred with this judgment. However, the difference in programmability scores is very slight.

The benefit of using OpenMP was a much simpler and shorter syntax. However, the API required specific ordering and created deep nesting. The pthreads version required more pointer manipulation, especially in passing arguments; however the behavior was more explicit and easier to debug.

One downside to these metrics is the lack of a measurement of debugging or code creation time. Measuring these aspects reliably would require collecting development statistics for a team of programmers for each target system. For this particular example it is the programmer’s opinion that OpenMP would have fared better in a head-to-head evaluation.

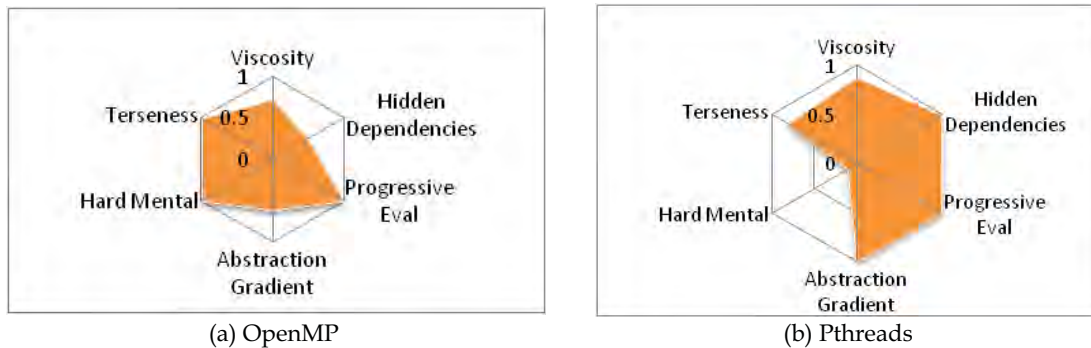


Figure 28. Cognitive dimension scores for two implementations of the Quicksort algorithm.

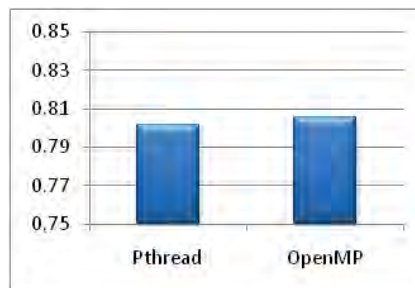


Figure 29. Programmability score for two Quicksort implementations.

Progressive evaluation was difficult to measure for this example, since the only parallel function was called recursively. Consequently, replacing it with a stub would have resulted in a null program. This observation suggests that additional research is needed to define a robust method to score the progressive evaluation dimension.

6.5.2 Example 2 - MapReduce

Map Reduce aims to process large amounts of data in parallel by mapping subsets of the data to cores for manipulating and then reduces the processed subsets into a single massive data set. For this experiment, a MapReduce algorithm was created that read in 1024 files with "links." Each link specified a source node (an integer value) and a destination node (an integer value). The MapReduce algorithm creates a data structure that is keyed by the destination node that points to a list of all sources that point to that destination.

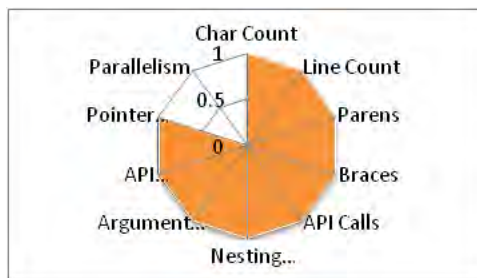
The implementation consisted of two files: the main file with the map/reduce functions, and `sort.cpp` which is the serial version of the quicksort algorithm from example 1. Since `sort.cpp` was a constant file, only the main file was evaluated.

It was found that the OpenMP version of the parallel code was much easier to program. OpenMP required only two lines of code to be added to the serial code. Pthreads required many more lines to be added, and required all functions to be modified.

Table 15 gives the basic cognitive dimension measurements for the two MapReduce programs. Figure 30 and Figure 31 show the Kiviat diagrams and overall programmability score for this example. The OpenMP version scored very high on all dimensions except parallelism. The Pthreads implementation was able to more easily extract parallelism than OpenMP. Beyond 4 threads, OpenMP would segmentation fault on the Macintosh laptop. Pthreads was able to spawn all 1024 threads in parallel for mapping and reducing. This suggests that a programmability metric must be able to take into account the parallelism achieved. For instance, if the programmability score was scaled by the parallelism score, the Pthreads version would score as 147x better than OpenMP.

Table 15. Measurements for MapReduce program.

	Pthreads	OpenMP
Char Count	3221	2642
Line Count	183	149
Parentheses	61	50
Braces	22	18
API Calls	5	2
Nesting Level	1	1
Argument Count	12	6
API Ordering Constraints	2	0
Pointer Manipulation Count	14	0
Parallelism	1024	4



(a) OpenMP



(b) Pthreads

Figure 30. Cognitive dimension scores for two implementations of the MapReduce algorithm.

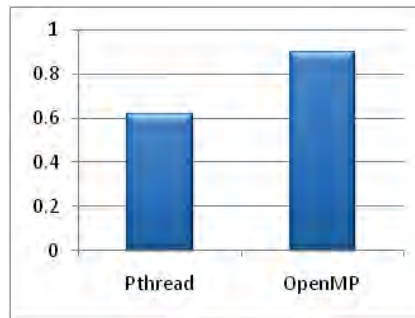


Figure 31. Programmability score for two MapReduce implementations.

6.6 Programmability Metric Conclusions

The proposed methodology for measuring programmability requires significant additional research to determine if it can serve as a practical basis for a programmability metric for high performance, large scale UHPC and EES systems. Additional work is needed in the concept development; selection and definition of programming problems; scoring methodology; and application to large-scale applications and machines.

Regarding the basic concept and definition of the programmability metric, further investigation is needed to establish the degree of correlation between programmability and cognitive dimensions, since the existence of such a correlation is the fundamental assumption of the proposed methodology. It is expected that a review of HCI research should be useful.

Regarding the selection of programming problems, attention is needed to ensure adequate coverage of important application areas by the basis set of canonical parallel patterns. While use of the Berkeley canonical problems is currently proposed, these are not necessarily exhaustive of all the parallel patterns that should be tested, nor representative of all application areas of importance to the DoD. In addition to pattern and application coverage, the criteria for problem selection should include large amounts of accessible concurrency and an objective and relatively easy means of validation of the developers' program.

Regarding scoring methodology, additional work is needed on automating scoring of the cognitive dimensions from the parallel codes. Furthermore, means to specify and score the codes that ensure performance and programmability are targeted jointly are important.

Experiments to date have been restricted to just two toy-size problems on a laptop computer. While GT believes that the concepts proposed appear applicable to large-scale machines, there are undoubtedly a number of issues that will arise as problem and machine sizes are scaled up. It is important for the evaluator to create code specifications that encourage a specific amount of parallelism and performance.

REFERENCES

- [1] Defense Advanced Research Projects Agency (DARPA), www.darpa.mil.
- [2] DARPA High Productivity Computing Systems web site, [http://www.darpa.mil/Our_Work/I2O/Programs/High_Productivity_Computing_Systems_\(HPCS\).aspx](http://www.darpa.mil/Our_Work/I2O/Programs/High_Productivity_Computing_Systems_(HPCS).aspx).
- [3] "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems", P. Kogge, editor, September 28, 2008. Available at users.ece.gatech.edu/~mrichard/ExascaleComputingStudyReports/ECS_reports.htm.
- [4] "ExaScale Computing Software Study: Software Challenges in Extreme Scale Systems", V. Sarkar, editor, September 14, 2009. Available at users.ece.gatech.edu/~mrichard/ExascaleComputingStudyReports/ECS_reports.htm.
- [5] "System Resilience at Extreme Scale", E. Elnozahy, editor, September 14, 2009. Available at <http://institute.lanl.gov/resilience/docs/>.
- [6] Richards, M. A., Gadiant, A., Frank, G. A., and Harr, R., "The RASSP Program: Origin, Concepts, and Status - An Introduction to the Issue", *J. VLSI Sig. Proc. Sys. For Signal, Image, and Video Tech.*, vol. 15, nos. 1/2, pp. 7-28, Jan. 1997.
- [7] Boehm B, "A Spiral Model of Software Development and Enhancement," *ACM SIGSOFT Software Engineering Notes*, J. ACM, 11(4):14-24, August 1986.
- [8] Gajski, D. D. and Kuhn, R. H., "Guest Editor's Introduction - New VLSI Tools", *IEEE Computer*, vol. 16(12), pp. 11-14, 1983.
- [9] Chung, E. S., Nurvitadhi, E., Hoe, J. C., Falsafi, B., and Mai, K., "A complexity-effective architecture for accelerating full-system multiprocessor simulations using FPGAs", *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, Monterey, California, 2008.
- [10] Mahlke, S. A, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," 25th International Symposium on Microarchitecture, 1992.
- [11] Bryan, P. D., Rosier, M. C. Conte, T. M. "Reverse State Reconstruction for Sampled Microarchitectural Simulation," 2007 IEEE Intl. Symp. on Performance Analysis of Systems and Software (ISPASS), April 2007.
- [12] Kessler, R. E., Hill, M. D., and Wood, D. A., "A Comparison of Trace-sampling Techniques for Multi-megabyte Caches," *IEEE Trans. Computing*, vol. C-43, June 1994.
- [13] Wood, D. A., Hill, M. D., and Kessler, R. E., "A Model for Estimating Trace-sample Miss Ratios," *Proc. ACM SIGMETRICS 1991 Conf. on Measurement and Modeling of Comput. Sys.*, May 1991.

- [14] Liu, W. and Huang, M. C., "EXPERT: Expedited Simulation Exploiting Program Behavior Repetition," ICS, 2004.
- [15] Conte, T. M., Hirsch, M. A., and Menezes, K. N., "Reducing State Loss for Effective Trace Sampling of Superscalar Processors," IEEE Intl. Conf. Computer Design, 1996.
- [16] Bryan, P. D., Conte, T. M., "Combining Cluster Sampling with Single Pass Methods for Efficient Sampling Regimen Design," IEEE Intl. Conf. Computer Design, 2007.
- [17] Fu, J. W. C., and Patel, J. H., "Trace Driven Simulation using Sampled Traces," *Proc. 27th Hawaii Int'l. Conf. on System Sciences*, (Maui, HI), Jan. 1994.
- [18] Iyengar, V. S., Trevillyan, L. H., and Bose, P., "Representative Traces for Processor Models with Infinite Cache", IEEE Intl. Symp. High Performance Computer Architecture, 1996.
- [19] Lui, L., and Peir, J., "Cache Sampling by Sets," *IEEE Trans. VLSI Systems*, vol. 1, June 1993.
- [20] Student, A. [William Sealy Gosset], "The probable error of a mean," *Biometrika*, March 1908, pp 1-25.
- [21] Haskins, J. W., and Skadron, K., "Memory Reference Reuse Latency: Accelerated Sampled Microarchitecture Simulation," IEEE Intl. Symp. on Performance Analysis of Systems and Software (ISPASS), 2003.
- [22] Sherwood, T., Perelman, E., Hamerly, G., and Calder, B., "Automatically Characterizing Large Scale Program Behavior," Intl. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2002.
- [23] W. L. Melvin and G. A. Showman, "Performance results for a knowledge-aided clutter mitigation architecture", 2007 IET Intl. Conf. Radar Systems, pp. 1-5, 2007.
- [24] Top 500 list for November 2010. <http://www.top500.org/lists/2010/11>.
- [25] Ubiquitous High Performance Computing (UHPC) Program web site, [http://www.darpa.mil/Our_Work/I2O/Programs/Ubiquitous_High_Performance_Computing_\(UHPC\).aspx](http://www.darpa.mil/Our_Work/I2O/Programs/Ubiquitous_High_Performance_Computing_(UHPC).aspx).
- [26] "Supercomputer hopes to say Gotcha to future terrorists", USAF press release, Sept. 30, 2009. Available at <http://www.af.mil/news/story.asp?id=123170460>.
- [27] S. M. Scarborough et al, "A challenge problem for SAR-based GMTI in urban environments", *Proc. SPIE Vol. 7337, 73370G* (Apr. 28, 2009).
- [28] M. Soumekh, *Synthetic Aperture Radar Signal Processing with MATLAB Algorithms*. Wiley, New York, 1999.
- [29] A. Goshtasby, "Registration of images with geometric distortions", *IEEE Trans. Geosci. Remote Sensing*, vol. 26, no. 11, pp. 60-64, Jan. 1988.

- [30] F. L. Bookstein, "Principal warps: Thin-plate splines and the decomposition of deformations", *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 11, no. 6, pp. 567-585, Jun. 1989.
- [31] C. D. Austin, E. Ertin, and R. L. Moses, "Sparse multipass 3D SAR imaging: applications to the GOTCHA data set", *Proc. SPIE* Vol. 7337, 733703 (Apr. 28, 2009).
- [32] DARPA Urban Challenge web site. <http://archive.darpa.mil/grandchallenge/>.
- [33] "Team Sting" technical paper, DARPA Urban Challenge, June 1 2007. Available at http://archive.darpa.mil/grandchallenge/TechPapers/Sting_Racing.pdf.
- [34] D. Wooden et al, "A Modular, Hybrid System Architecture for Autonomous, Urban Driving", *J. Aerospace Computing, Info., and Comm.*, Vol. 4, pp. 1047-1058, December 2007.
- [35] Z. Kim, "Robust Lane Detection and Tracking in Challenging Scenarios", *IEEE Trans. Intelligent Transportation Systems*, vol. 9 (1), pp. 16-26, 2008.
- [36] M. A. Fischler and R. C. Bolles, "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography", *Comm. of the ACM*, 24: 381-395, June 1981.
- [37] M. Montemerlo and S. Thrun, *FastSLAM*. Springer, 2007.
- [38] J. Weinberg, M. O. McCracken, A. Snavey, E. Strohmaier, "Quantifying Locality In The Memory Access Patterns of HPC Applications", *Supercomputing 2005*. November 2005, Seattle, WA.
- [39] C. Ding and Y. Zhong, "Predicting whole-program locality with reuse distance analysis", *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.
- [40] A. Oppenheim and R. Schaffer. *Discrete-Time Signal Processing*. Prentice Hall, 1999.
- [41] G. Golub and C. Van Loan. *Matrix Computations*. The John Hopkins University Press, 1996.
- [42] Vector, Signal, Image Processing Library (VSIPL) web site. <http://www.vsipl.org/>.
- [43] High Performance Embedded Computing Software Initiative (HPEC-SI) web site. <http://www.hpec-si.org/>.
- [44] GPU-VSIPL web site. <http://gpu-vsipl.gtri.gatech.edu/>.
- [45] A. R. Kerr, D. P. Campbell, and M. A. Richards, "GPU VSIPL: High-Performance VSIPL Implementation for GPUs", *Proceedings 2008 High Performance Embedded Computing Workshop*, MIT Lincoln Laboratory, September 23-25, 2008. Available at <http://www.ll.mit.edu/HPEC/agendas/proc08/agenda.html>.

- [46] M. A. Richards, *Fundamentals of Radar Signal Processing*, McGraw-Hill, New York, 2005.
- [47] VSIPL Test Suite, software available at <http://www.vsipl.org>.
- [48] A. Blackwell, "Cognitive Dimensions of Notations," in *Visual Languages and Human-Centric Computing*, 2005 IEEE Symp. on Visual Languages and Human-Centric Comp., 2005, page 3.
- [49] T. Mattson and M. Wrinn, "Parallel programming: Can we PLEASE get it right this time?" in *Proc. 45th ACM/IEEE Design Automation Conference*, 2008, pp. 7-11.
- [50] T. G. S. Mattson, Beverly A.; Massingill, Berna L., *Patterns for Parallel Programming*. Addison-Wesley, 2004.
- [51] J. L. Ortega-Arjona, *Patterns for Parallel Software Design*. West Sussex: Wiley, 2010.
- [52] *A Pattern Language for Parallel Programming*, ver2.0. Available: at <http://parlab.eecs.berkeley.edu/wiki/patterns/patterns>.
- [53] K. Asanovic *et al*, "A view of the parallel computing landscape", *Comm. ACM*, vol. 52, no. 10, Oct. 2009.